Programming with OpenGL: Advanced T e c h n i q u e s

Organizer:
Tom McReynolds
Silicon Graphics

May 1, 1997

SIGGRAPH ‘97 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 helps students achieve two goals: they gain a deeper insight into OpenGL functionality and computer graphics concepts, while expanding their “toolbox” of useful OpenGL techniques.

Speakers

David Blythe

David Blythe is a Principal Engineer with the Advanced Systems Division at Silicon Graphics. David joined SGI in 1991 and has contributed to the development of RealityEngine and InfiniteReality graphics. 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. David received both a B.S. and M.S. degree in computer science from the University of Toronto.
Email: blythe@asd.sgi.com

Celeste Fowler

Celeste Fowler is a software engineer in the Advanced Systems Division at Silicon Graphics. She worked on the OpenGL imaging pipeline for the InfiniteReality graphics system and on the OpenGL display list implementation for InfiniteReality and RealityEngine.

Before coming to SGI, Celeste attended Princeton University where she did research on radiosity techniques and TA'd courses in computer graphics and programming systems.
Email: celeste@asd.sgi.com

Brad Grantham

Brad Grantham currently contributes to the design and implementation of Silicon Graphics' high-level graphics toolkits, including OpenGL++, a scene graph toolkit for OpenGL. Brad previously worked on the Windows 95 port and Java bindings for Cosmo 3D and, before that, worked in the IRIS Performer group.

Before joining SGI, Brad wrote UNIX kernel code and imaging codecs. He received a B.S. from Virginia Tech in 1992, and his previous claim to fame is MacBSD, BSD UNIX for the Macintosh.
Email: grantham@sgi.com

Simon Hui

Simon Hui is a software engineer at 3Dfx Interactive, Inc. He currently works on OpenGL and other graphics libraries for PC and consumer platforms.

Prior to joining 3Dfx, Simon worked on IRIS Performer, a realtime graphics

toolkit, in the Advanced Systems Division at Silicon Graphics. He has also worked on OpenGL implementations for the RealityEngine and InfiniteReality. Simon received a B.A. in Computer Science from the University of California at Berkeley.
Email: simon@3dfx.com

Tom McReynolds

Tom McReynolds is a software engineer in the Core Rendering group at Silicon Graphics. He's implemented OpenGL extensions and done OpenGL performance work. He currently works on IRIS Performer, a real-time visualization library that uses OpenGL.

Prior to SGI, he worked at Sun Microsystems, where he developed graphics hardware support software and graphics libraries, including XGL.

Tom is also an adjunct professor at Santa Clara University, where he teaches courses in computer graphics using the OpenGL library. He has also presented at the X Technical Conference, SIGGRAPH '96, and SGI's 1996 Developer Forum.
Email: tomcat@asd.sgi.com

Paula Womack

Paula Womack manages the OpenGL group at Silicon Graphics. She is also a member of the OpenGL Architectural Review Board (the OpenGL ARB) which is responsible for defining and enhancing OpenGL.

Prior to joining Silicon Graphics, Paula worked on OpenGL at Kubota and Digital Equipment. She has a B.S. in Computer Engineering from the University of California at San Diego.
Email: womack@asd.sgi.com

Contents

List of Figures

1 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 easy to understand, but its simplicity and orthogonality enable a multitude of interesting effects. The goal of this course is to demonstrate how to generate more satisfying images using OpenGL. There are three general areas of discussion: generating aesthetically pleasing or realistic looking basic images, computing interesting effects, and generating more sophisticated images.

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 the most recent version of OpenGL, version 1.1, 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 use terminology and notation consistent with other OpenGL documentation.

1.1 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 like to thank some of our more fruitful and patient sources of material: John Airey, Remi Arnaud, Brian Cabral, Bob Drebin, Phil Lacroute, Mark Peercy, and David Yu.

Credit should also be given to our army of reviewers: John Airey, Allen Akin, Brian Cabral, Tom Davis, Bob Drebin, Ben Garlick, Michael Gold, Robert Grzeszczuk, Paul Haeberli, Michael Jones, Phil Keslin, Phil Lacroute, Erik Lindholm, Mark Peercy, Mark Young, David Yu, and particularly Mark Segal for having the endurance to review for us two years in a row.

We would like to acknowledge Atul Narkhede and Rob Wheeler for coding prototype algorithms, and Chris Everett for once again providing his invaluable production expertise and assistance this year, and Dany Galgani for some really nice illustrations.

We would also like to thank John Airey, Paul Heckbert, Phil Lacroute, Mark

Segal, Michael Teschner, and Tim Wiegand for providing material for inclusion in the reprints section.

Permission to reproduce [50] has been granted by Computer Graphics Forum.

Once again this year the IRIS Performer Team receives our gratitude for "covering" for two of us while these notes were being written.

1.2 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_sig97.html

2 About OpenGL

Before getting into the intricacies of using OpenGL, we begin with 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 programmer 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 systems such as VRML [9] are descriptive; one simply specifies that a red sphere should be drawn at certain coordinates. 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 trade-off 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. Think of OpenGL as a "graphics assembly language": the pieces of OpenGL functionality can be combined as building blocks to create innovative techniques and produce new graphics capabilities.

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 exactly the same image. This allows 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; limiting 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 diverse set of machines.

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 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. This may result in either the use of different algorithms being invoked or arithmetic with different precision than that used in the hardware accelerator. In such a case, when texturing is enabled, a slightly different set of pixels in the window may be written 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.

3 Modelling

Rendering is only half the story. Great computer graphics starts with great images and geometric models. This section describes some modelling does and don'ts, and describes a high performance way of performing CSG operations.

3.1 Modelling Considerations

OpenGL is a renderer not a modeller. There are utility libraries such as the OpenGL Utility Library (GLU) which can assist with modelling tasks, but for all practical purposes is the application's responsibility. Attention to modelling considerations is important; the image quality is directly related to the quality of the modelling. For example, undertessellated geometry produces poor silhouette edges. Other artifacts result from a combination of the model and OpenGL's ordering scheme. For example, interpolation of colors determined as a result of evaluation of a lighting equation at the vertices can result in a less than pleasing specular highlight 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 triangle fans. Primitives such as polygons and quads are usually decomposed by OpenGL into triangles before rasterization. OpenGL does not provide controls over how this decomposition is done, so for more predictable results, the application should do the tessellation directly. Application tessellation is also more efficient if the

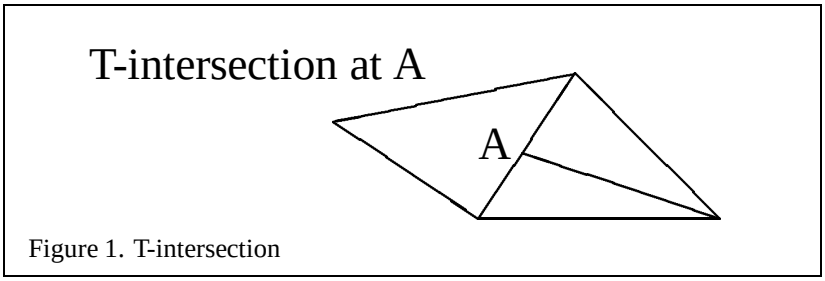


Figure 1. T-intersection

same model is to be drawn multiple times (e.g., multiple instances per frame, as part of a multipass algorithm, or for multiple frames). The second release of the GLU library (version 1.1) includes a very good general polygon tessellator; 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 (Figure 1).

   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 positions so that the edge equations are the same.

   Note that this requirement must be satisfied 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. This often requires adding edges and creating new triangles to "stitch" the edges of abutting objects together seamlessly.

3. The T-intersection problem has consequences for 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, applications tessellate the object to varying degrees depending on the area of the screen that it covers. This ensures that time is not wasted drawing many triangles that cover only a few

pixels on the screen. This is a difficult mechanism to implement correctly; if the view of the object is changing the changes in tessellation from frame to frame may result in noticeable motion artifacts. Often it is best to either undertessellate and live with those artifacts or overtessellate and accept reduced performance. The GLU NURBS library is and example of a package which implements view-dependent tessellation 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 The vertex at 0 must be the same as the one at 2π. Note that the OpenGL specification is very strict in this regard as even the `glMapGrid` routine must evaluate 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 at least two reasons for maintaining this consistency. First the OpenGL face culling method can be used as an efficient form of hidden surface elimination for convex surfaces and, second, several algorithms can exploit the ability to selectively draw only the frontfacing or backfacing polygons of a surface.

3.2 Decomposition and Tessellation

Tessellation refers to the process of decomposing a complex surface such as a sphere into simpler primitives such as triangles or quadrilaterals. Most OpenGL implementations are tuned to process triangle strips and triangle fans efficiently. Triangles are desirable because they are planar, easy to rasterize, and can always be interpolated unambiguously. When an implementation is optimized for processing

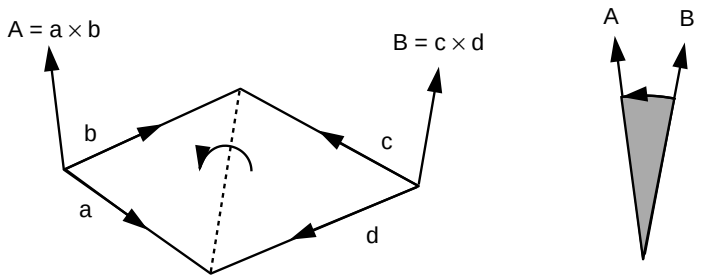


Figure 2. Quadrilateral decomposition

triangles, more complex primitives such as quad strips, quads, and polygons are decomposed into triangles early in the pipeline.

If the underlying implementation is performing this decomposition, there is a performance benefit in performing this decomposition a priori, either when the database is created or at application initialization time, rather than each time the primitive is issued. A second advantage of performing this decomposition under the control of the application is that the decomposition can be done consistently and independently of the OpenGL implementation. Since OpenGL doesn't specify its decomposition algorithm, different implementations may decompose a given quadrilateral along different diagonals. This can result in an image that is shaded differently and has different silhouette edges.

Quadrilaterals are decomposed by finding the diagonal that creats two triangles with the greatest difference in orientation. A good way to find this diagonal is to compute the angles between the normals at opposing vertices, compute the dot product, then choose the pair with the largest angle (smallest dot product) as shown in Figure 2. The normals for a vertex can be computed by taking the cross products of the the two vectors with origins at that vertex.

Tessellation of simple surfaces such as spheres and cylinders is not difficult. Most implementations of the GLU library use a simple lattitude-longitude tessellation for a sphere. While the algorithm is simple to implement, it has the disadvantage that the triangles produced from the tessellation have widely varying sizes. These widely varying sizes can cause noticeable artifacts, particularly if the object is lit and rotating.

A better algorithm generates triangles with sizes that are more consistent. Octahedral and Icosahedral tessellations work well and are not very difficult to implement. An octahedral tessellation approximates a sphere with an octahedron whose

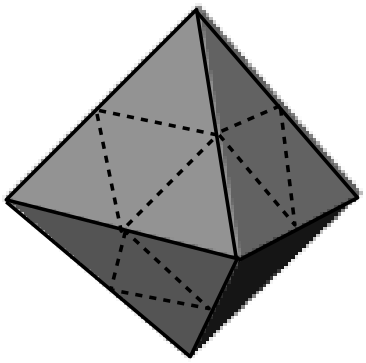


Figure 3. Octahedron with triangle subdivision

vertices are all on the unit sphere. Since the faces of the octahedron are triangles they can easily be split into 4 triangles, as shown in Figure 3.

Each triangle is split by creating a new vertex in the middle of each edge and adding three new edges. These vertices are scaled onto the unit sphere by dividing them by their distance from the origin (normalizing them). This process can be repeated as desired, recursively dividing all of the triangles generated in each iteration.

The same algorithm can be applied using an icosahedron as the base object, recursively dividing all 20 sides. In both cases the algorithms can be coded so that triangle strips are generated instead of independent triangles, maximizing rendering performance.

3.3 Capping Clipped Solids with the Stencil Buffer

When dealing with solid objects it is often useful to clip the object against a plane and observe the cross section. OpenGL's user-defined clipping planes allow an application to clip the scene by a plane. The stencil buffer provides an easy method for adding a "cap" to objects that are intersected by the clipping plane. A capping polygon is embedded in the clipping plane and the stencil buffer is used to trim the polygon to the interior of the solid.

For more information on the techniques using the stencil buffer, see Section 14.

If some care is taken when constructing the object, solids that have a depth complexity greater than 2 (concave or shelled objects) and less than the maximum value

of the stencil buffer can be rendered. Object surface polygons must have their vertices ordered so that they face away from the interior for face culling purposes.

The stencil buffer, color buffer, and depth buffer are cleared, and color buffer writes are disabled. The capping polygon is rendered into the depth buffer, then depth buffer writes are disabled. The stencil operation is set to increment the stencil value where the depth test passes, and the model is drawn with `glCullFace(GL_BACK)`. The stencil operation is then set to decrement the stencil value where the depth test passes, and the model is drawn with `glCullFace(GL_FRONT)`.

At this point, the stencil buffer is 1 wherever the clipping plane is enclosed by the frontfacing and backfacing surfaces of the object. The depth buffer is cleared, color buffer writes are enabled, and the polygon representing the clipping plane is now drawn using whatever material properties are desired, with the stencil function set to `GL_EQUAL` and the reference value set to 1. This draws the color and depth values of the cap into the framebuffer only where the stencil values equal 1.

Finally, stenciling is disabled, the OpenGL clipping plane is applied, and the clipped object is drawn with color and depth enabled.

3.4 Constructive Solid Geometry with the Stencil Buffer

Before continuing, the it may help for the reader to be familiar with the concepts presented in Section 14.

Constructive solid geometry (CSG) models are constructed through the intersection ($\cap$), union ($\cup$), and subtraction ($-$) of solid objects, some of which may be CSG objects themselves[17]. The tree formed by the binary CSG operators and their operands is known as the CSG tree. Figure 4 shows an example of a CSG tree and the resulting model.

The representation used in CSG for solid objects varies, but we will consider a solid to be a collection of polygons forming a closed volume. “Solid”, “primitive”, and “object” are used here to mean the same thing.

CSG objects have traditionally been rendered through the use of ray-casting, which is slow, or through the construction of a boundary representation (B-rep).

B-reps vary in construction, but are generally defined as a set of polygons that form the surface of the result of the CSG tree. One method of generating a B-rep is to take the polygons forming the surface of each primitive and trimming away the polygons (or portions thereof) that don’t satisfy the CSG operations. A B-rep models are typically generated once and then manipulated as a static model because they are slow to generate.

Drawing a CSG model using stencil usually means drawing more polygons than a B-rep would contain for the same model. Enabling stencil also may reduce perfor-

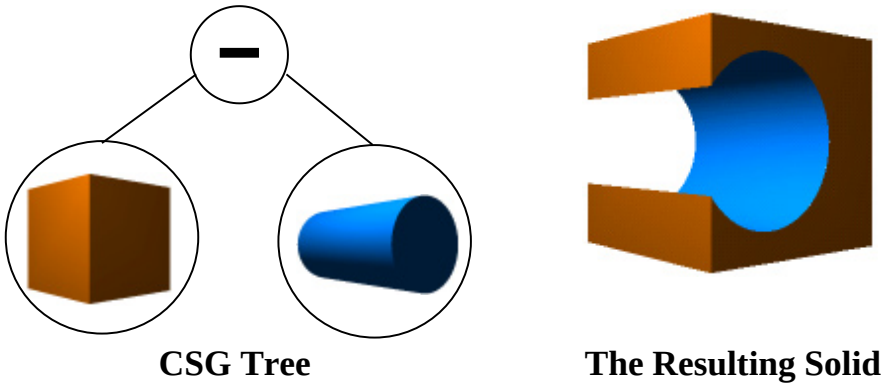


Figure 4. An Example Of Constructive Solid Geometry

mance. Nonetheless, some portions of a CSG tree may be interactively manipulated using stencil if the remainder of the tree is cached as a B-rep.

The algorithm presented here is from a paper by Tim F. Wiegand describing a GL-independent method for using stencil in a CSG modelling system for fast interactive updates. The technique can also process concave solids, the complexity of which is limited by the number of stencil planes available. A reprint of Wiegand's paper is included in the Appendix.

The algorithm presented here assumes that the CSG tree is in "normal" form. A tree is in normal form when all intersection and subtraction operators have a left subtree which contains no union operators and a right subtree which is simply a primitive (a set of polygons representing a single solid object). All union operators are pushed towards the root, and all intersection and subtraction operators are pushed towards the leaves. For example, $(((A \cap B) - C) \cup (((D \cap E) \cap G) - F)) \cup H$ is in normal form; Figure 5 illustrates the structure of that tree and the characteristics of a tree in normal form.

A CSG tree can be converted to normal form by repeatedly applying the following set of production rules to the tree and then its subtrees:

1. $X - (Y \cup Z) \rightarrow (X - Y) - Z$
2. $X \cap (Y \cup Z) \rightarrow (X \cap Y) \cup (X \cap Z)$
3. $X - (Y \cap Z) \rightarrow (X - Y) \cup (X - Z)$
4. $X \cap (Y \cap Z) \rightarrow (X \cap Y) \cap Z$

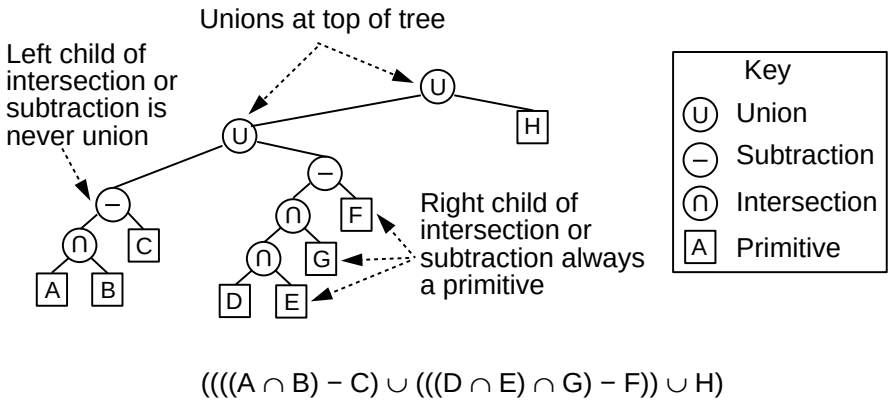


Figure 5. A CSG tree in normal form

5. $X - (Y - Z) \rightarrow (X - Y) \cup (X \cap Z)$
6. $X \cap (Y - Z) \rightarrow (X \cap Y) - Z$
7. $(X - Y) \cap Z \rightarrow (X \cap Z) - Y$
8. $(X \cup Y) - Z \rightarrow (X - Z) \cup (Y - Z)$
9. $(X \cup Y) \cap Z \rightarrow (X \cap Z) \cup (Y \cap Z)$

X, Y, and Z here match both primitives or subtrees. Here's the algorithm used to apply the production rules to the CSG tree:

```
normalize(tree *t)
{
    if(isPrimitive(t))
        return;

    do{
        while(matchesRule(t)) /* Using rules given above */
            applyFirstMatchingRule(t);
        normalize(t->left);
    }while( ! (isUnionOperation(t) ||
         (isPrimitive(t->right) &&
```

```
          ! isUnionOperation(T->left))));
    normalize(t->right);
}
```

Normalization may increase the size of the tree and add primitives which do not contribute to the final image. The bounding volume of each CSG subtree can be used to prune the tree as it is normalized. Bounding volumes for the tree may be calculated using the following algorithm:

```
findBounds(tree *t)
{
    if(isPrimitive(t))
        return;

    findBounds(t->left);
    findBounds(t->right);

    switch(t->operation){
        case union:
            t->bounds = unionOfBounds(t->left->bounds,
                                      t->right->bounds);
        case intersection:
            t->bounds = intersectionOfBounds(t->left->bounds,
                                             t->right->bounds);
        case subtraction:
            t->bounds = t->left->bounds;
    }
}
```

CSG subtrees rooted by the intersection or subtraction operators may be pruned at each step in the normalization process using the following two rules:

1. if T is an intersection and not `intersects(T->left->bounds, T->right->bounds)`, delete T.
2. if T is a subtraction and not `intersects(T->left->bounds, T->right->bounds)`, replace T with T->left.

The normalized CSG tree is a binary tree, but it's important to think of the tree rather as a "sum of products" to understand the stencil CSG procedure.

Consider all the unions as sums. Next, consider all the intersections and subtractions as products. (Subtraction is equivalent to intersection with the complement of

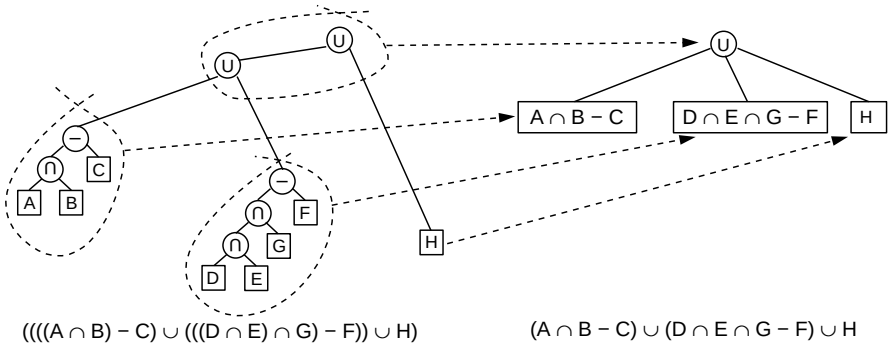


Figure 6. Thinking of a CSG tree as a sum of products

the term to the right. For example, $A - B = A \cap \bar{B}$.) Imagine all the unions flattened out into a single union with multiple children; that union is the "sum". The resulting subtrees of that union are all composed of subtractions and intersections, the right branch of those operations is always a single primitive, and the left branch is another operation or a single primitive. You should read each child subtree of the imaginary multiple union as a single expression containing all the intersection and subtraction operations concatenated from the bottom up. These expressions are the "products". For example, you should think of $((A \cap B) - C) \cup (((G \cap D) - E) \cap F) \cup H$ as meaning $(A \cap B - C) \cup (G \cap D - E \cap F) \cup H$. Figure 6 illustrates this process.

At this time redundant terms can be removed from each product. Where a term subtracts itself ($A - A$), the entire product can be deleted. Where a term intersects itself ($A \cap A$), that intersection operation can be replaced with the term itself.

All unions can be rendered simply by finding the visible surfaces of the left and right subtrees and letting the depth test determine the visible surface. All products can be rendered by drawing the visible surfaces of each primitive in the product and trimming those surfaces with the volumes of the other primitives in the product. For example, to render $A - B$, the visible surfaces of A are trimmed by the complement of the volume of B, and the visible surfaces of B are trimmed by the volume of A.

The visible surfaces of a product are the front facing surfaces of the operands of intersections and the back facing surfaces of the right operands of subtraction. For example, in $(A - B \cap C)$, the visible surfaces are the front facing surfaces of A and C, and the back facing surfaces of B.

Concave solids are processed as sets of front or back facing surfaces. The "convexity" of a solid is defined as the maximum number of pairs of front and back sur-

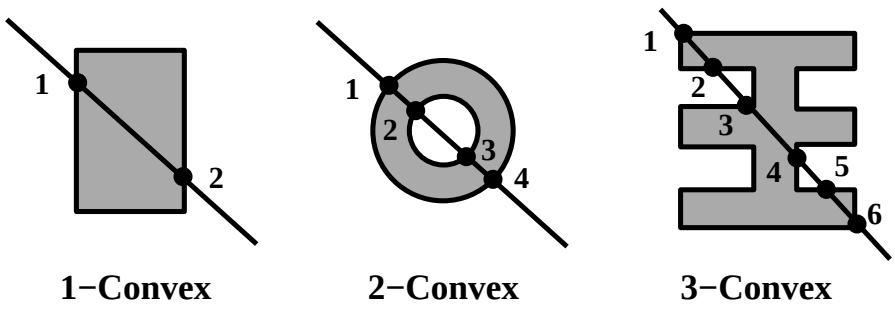


Figure 7. Examples of *n*-convex solids

faces that can be drawn from the viewing direction. Figure 7 shows some examples of the convexity of objects. The *n*th front surface of a *k*-convex primitive is denoted A_{nf}, and the *n*th back surface is A_{nb}. Because a solid may vary in convexity when viewed from different directions, accurately representing the convexity of a primitive may be difficult and may also involve reevaluating the CSG tree at each new view. Instead, the algorithm must be given the *maximum possible* convexity of a primitive, and draws the *n*th visible surface by using a counter in the stencil planes.

The CSG tree must be further reduced to a "sum of partial products" by converting each product to a union of products, each consisting of the product of the visible surfaces of the target primitive with the remaining terms in the product.

For example, if A, B, and D are 1-convex and C is 2-convex:

$$\begin{aligned}
(A - B \cap C \cap D) \quad &\rightarrow \\
(A_{0f} - B \cap C \cap D) \quad &\cup \\
(B_{0b} \cap A \cap C \cap D) \quad &\cup \\
(C_{0f} \cap A - B \cap D) \quad &\cup \\
(C_{1f} \cap A - B \cap D) \quad &\cup \\
(D_{0f} \cap A \cap B \cap C) \quad &
\end{aligned}$$

Because the target term in each product has been reduced to a single front or back facing surface, the bounding volumes of that term will be a subset of the bounding volume of the original complete primitive. Once the tree is converted to partial products, the pruning process may be applied again with these subset volumes.

In each resulting child subtree representing a partial product, the leftmost term is called the "target" surface, and the remaining terms on the right branches are called "trimming" primitives.

The resulting sum of partial products reduces the rendering problem to rendering each partial product correctly before drawing the union of the results. Each partial product is rendered by drawing the target surface of the partial product and then "classifying" the pixels generated by that surface with the depth values generated by each of the trimming primitives in the partial product. If pixels drawn by the trimming primitives pass the depth test an even number of times, that pixel in the target primitive is "out", and discarded. If the count is odd, the target primitive pixel is "in"', and kept.

Because the algorithm saves depth buffer contents between each object, we optimize for depth saves and restores by drawing as many of target and trimming primitives for each pass as we can fit in the stencil buffer.

The algorithm uses one stencil bit (S_p) as a toggle for trimming primitive depth test passes (parity), *n* stencil bits for counting to the *n*th surface (S_{count}), where *n* is the smallest number for which 2^n is larger than the maximum convexity of a current object, and as many bits are available (S_a) to accumulate whether target pixels have to be discarded. Because S_{count} will require the `GL_INCR` operation, it must be stored contiguously in the least-significant bits of the stencil buffer. S_p and S_{count} are used in two separate steps, and so may share stencil bits.

For example, drawing 2 5-convex primitives would require 1 S_p bit, 3 S_{count} bits, and 2 S_a bits. Because S_p and S_{count} are independent, the total number of stencil bits required would be 5.

Once the tree has been converted to a sum of partial products, the individual products are rendered. Products are grouped together so that as many partial products can be rendered between depth buffer saves and restores as the stencil buffer has capacity.

For each group, writes to the color buffer are disabled, the contents of the depth buffer are saved, and the depth buffer is cleared. Then, every target in the group is classified against its trimming primitives. The depth buffer is then restored, and every target in the group is rendered against the trimming mask. The depth buffer save/restore can be optimized by saving and restoring only the region containing the screen-projected bounding volumes of the target surfaces.

```
for each group
    glReadPixels(...);
    classify the group
    glStencilMask(0); /* so DrawPixels won't affect Stencil */
    glDrawPixels(...);
```

```
    render the group
```

Classification consists of drawing each target primitive's depth value and then clearing those depth values where the target primitive is determined to be outside the trimming primitives.

```
glClearDepth(far);
glClear(GL_DEPTH_BUFFER_BIT);
a = 0;
for each target surface in the group
    for each partial product targeting that surface
        render the depth values for the surface
        for each trimming primitive in that partial product
            trim the depth values against that primitive
        set Sa to 1 where Sa = 0 and Z < Zfar;
        a++;
```

The depth values for the surface are rendered by drawing the primitive containing the the target surface with color and stencil writes disabled. (S_{count}) is used to mask out all but the target surface. In practice, most CSG primitives are convex, so the algorithm is optimized for that case.

```
if(the target surface is front facing)
    glCullFace(GL_BACK);
else
    glCullFace(GL_FRONT);

if(the surface is 1-convex)
    glDepthMask(1);
    glColorMask(0, 0, 0, 0);
    glStencilMask(0);
    draw the primitive containing the target surface
else
    glDepthMask(1);
    glColorMask(0, 0, 0, 0);
    glStencilMask(Scount);
    glStencilFunc(GL_EQUAL, index of surface, Scount);
    glStencilOp(GL_KEEP, GL_KEEP, GL_INCR);
    draw the primitive containing the target surface
    glClearStencil(0);
    glClear(GL_STENCIL_BUFFER_BIT);
```

Then each trimming primitive for that target surface is drawn in turn. Depth testing is enabled and writes to the depth buffer are disabled. Stencil operations are masked to S_p and the S_p bit in the stencil is cleared to 0. The stencil function and operation are set so that S_p is toggled every time the depth test for a fragment from the trimming primitive succeeds. After drawing the trimming primitive, if this bit is 0 for uncomplemented primitives (or 1 for complemented primitives), the target pixel is "out", and must be marked "discard", by enabling writes to the depth buffer and storing the far depth value (Z_f) into the depth buffer everywhere that the S_p indicates "discard".

```
glDepthMask(0);
glColorMask(0, 0, 0, 0);
glStencilMask(mask for Sp);
glClearStencil(0);
glClear(GL_STENCIL_BUFFER_BIT);
glStencilFunc(GL_ALWAYS, 0, 0);
glStencilOp(GL_KEEP, GL_KEEP, GL_INVERT);
draw the trimming primitive
glDepthMask(1);
```

Once all the trimming primitives are rendered, the values in the depth buffer are Z_f for all target pixels classified as "out". The S_a bit for that primitive is set to 1 everywhere that the depth value for a pixel is not equal to Z_f, and 0 otherwise.

Each target primitive in the group is finally rendered into the frame buffer with depth testing and depth writes enabled, the color buffer enabled, and the stencil function and operation set to write depth and color only where the depth test succeeds and S_a is 1. Only the pixels inside the volumes of all the trimming primitives are drawn.

```
glDepthMask(1);
glColorMask(1, 1, 1, 1);
a = 0;
for each target primitive in the group
    glStencilMask(0);
    glStencilFunc(GL_EQUAL, 1, Sa);
    glCullFace(GL_BACK);
    draw the target primitive
    glStencilMask(Sa);
    glClearStencil(0);
    glClear(GL_STENCIL_BUFFER_BIT);
    a++;
```

There are further techniques described in [50] for adding clipping planes (half-spaces), including more normalization rules and pruning opportunities. This is especially important in the case of the near clipping plane in the viewing frustum.

A demo program showing complex CSG expressions rendered using the stencil buffer is on the website.

Source code for dynamically loadable Inventor objects implementing this technique is available at the Martin Center website at Cambridge, `http://www.arct.cam.ac.uk/mc/cadlab/inventor/`.

4 Geometry and Transformations

OpenGL has a simple and powerful transformation model. Since the transformation machinery in OpenGL is exposed in the form of the modelview and projection matrices, it's possible to develop novel uses for the transformation pipeline. This section describes some useful transformation techniques, and provides some additional insight into the OpenGL graphics pipeline.

4.1 Stereo Viewing

Stereo viewing is a common technique to increase visual realism or enhance user interaction with 3D scenes. Two views of a scene are created, one for the left eye, one for the right. Some sort of viewing hardware is used with the display, so each eye only sees the view created for it. The apparent depth of objects is a function of the difference in their positions from the left and right eye views. When done properly, objects appear to have actual depth, especially with respect to each other. When animating, the left and right back buffers are used, and must be updated each frame.

OpenGL supports stereo viewing, with left and right versions of the front and back buffers. In normal, non-stereo viewing, when not using both buffers, the default buffer is the left one for both front and back buffers. Since OpenGL is window system independent, there are no interfaces in OpenGL for stereo glasses, or other stereo viewing devices. This functionality is part of the OpenGL/Window system interface library; the style of support varies widely.

In order to render a frame in stereo:

- The display must be configured to run in stereo mode.
- The left eye view for each frame must be generated in the left back buffer.
- The right eye view for each frame must be generated in the right back buffer.

- The back buffers must be displayed properly, according to the needs of the stereo viewing hardware.

Computing the left and right eye views is fairly straightforward. The distance separating the two eyes, called the *interocular distance*, must be selected. Choose this value to give the proper size of the viewer's head relative to the scene being viewed. Whether the scene is microscopic or galaxy-wide is irrelevant. What matters is the size of the imaginary viewer relative to the objects in the scene. This distance should be correlated with the degree of perspective distortion present in the scene to produce a realistic effect.

4.1.1 Fusion Distance

The other parameter is the distance from the eyes where the lines of sight for each eye converge. This distance is called the *fusion distance*. At this distance objects in the scene will appear to be on the front surface of the display ("in the glass"). Objects farther than the fusion distance from the viewer will appear to be "behind the glass" while objects in front will appear to float in front of the display. The latter illusion is harder to maintain, since real objects visible to the viewer beyond the edge of the display tend to destroy the illusion.

Instead of assigning units to it, think of the fusion distance as a dimensionless quantity, relative to location of the front and back clipping planes. For example, you may want to set the fusion distance to be halfway between the front and back clipping planes. This way it is independent of the application's coordinate system, which makes it easier to position objects appropriately in the scene.

To model viewer attention realistically, the fusion distance should be adjusted to match the object in the scene that the viewer is looking at. This requires knowing where the viewer is looking. If head and eye tracking equipment is being used in the application finding the center of interest is straightforward. A more indirect approach is to have the user consciously designate the object being viewed. Clever psychology can sometimes substitute for eye tracking hardware. If the animated scene is designed in such a way as to draw the viewer's attention in a predictable way, or if the scene is very sparse, intelligent guesses can be made as to the viewers center of interest.

The view direction vector and the vector separating the left and right eye position are perpendicular to each other. The two view points are located along a line perpendicular to the direction of view and the "up" direction. The fusion distance is measured along the view direction. The position of the viewer can be defined to be at one of the eye points, or halfway between them. In either case, the left and right eye locations are positioned relative to it.

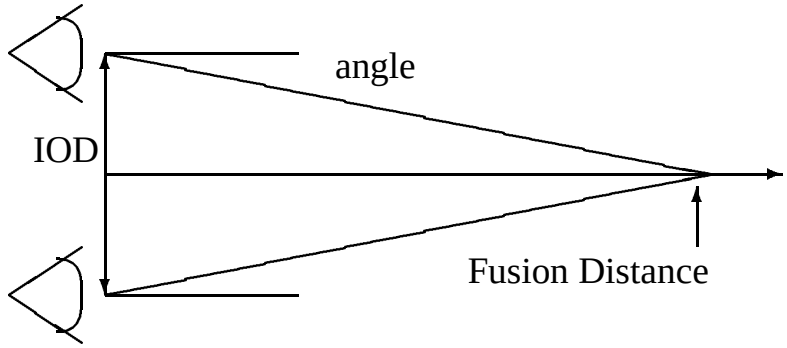


Figure 8. Stereo Viewing Geometry

If the viewer is taken to be halfway between the stereo eye positions, and assuming `gluLookAt` has been called to put the viewer position at the origin in eye space, then the fusion distance is measured along the negative Z axis (like the near and far clipping planes), and the two viewpoints are on either side of the origin along the X axis, at (-IOD/2, 0, 0) and (IOD/2, 0, 0).

4.1.2 Computing the Transforms

The transformations needed for stereo viewing are simple rotations and translations. Computationally, the stereo viewing transforms happen last, after the viewing transform has been applied to put the viewer at the origin. Since the matrix order is the reverse of the order of operations, the viewing matricies should be applied to the modelview matrix stack first.

The order of matrix operations should be:

1. Transform from viewer position to left eye view.
2. Apply viewing operation to get to viewer position (`gluLookAt` or equivalent).
3. Apply modeling operations.
4. Change buffers, repeat for right eye.

Assuming that the identity matrix is on the modelview stack:

```
glMatrixMode(GL_MODELVIEW);
```

```
glLoadIdentity(); /* the default matrix */
glPushMatrix()
glDrawBuffer(GL_BACK_LEFT)
glTranslatef(-IOD/2.f, 0, 0)
glRotatef(-angle, 0.f, 1.f, 0.f)
<viewing transforms>
<modeling transforms>
draw()
glPopMatrix();
glPushMatrix()
glDrawBuffer(GL_BACK_RIGHT)
glTranslatef(IOD/2, 0., 0.)
glRotatef(angle, 0.f, 1.f, 0.f)
<viewing transforms>
<modeling transforms>
draw()
glPopMatrix()
```

Where $angle$ is the inverse tangent of the ratio between the fusion distance and half of the interocular distance. $angle = \arctan(\frac{fusiondistance}{\frac{IOD}{2}})$ Each viewpoint is rotated towards the centerline halfway between the two viewpoints.

Another approach to implementing stereo transforms is to change the viewing tranform directly. Instead of adding an extra rotation and translation, use a separate call to `gluLookAt` for each eye view. Move fusion distance along the viewing direction from the viewer position, and use that point for the center of interest of both eyes. Translate the eye position to the appropriate eye, then render the stereo view for the corresponding buffer.

The difficulty with this technique is finding the left/right eye axis to translate along from the viewer position to find the left and right eye viewpoints. Since your now computing the left/right eye axis in object space, it is no longer constrained to be the X axis. Find the left/right eye axis in object space by taking the cross product of the direction of view and the up vector.

4.1.3 Rotate vs. Shear

Rotating the left and right eye view is is not the only way to generate the stereo images. The left and right eye views can be sheared instead. The left and eyes remain oriented along the direction of view, but each eyes view is sheared along z so that the two frustums converge at the fusion distance.

Although shearing each eye's view instead of rotating is less physically accurate, sheared stereo views can be easier for viewers to achieve stereo fusion. This is because the two eye views have the same orientation and lighting.

For objects that are far from the eye, the differences between the two approaches are small.

4.2 Depth of Field

Normal viewing tranforms act like a perfect pinhole camera; everything visible is in focus, regardless of how close or how far the objects are from the viewer. To increase realism, a scene can be rendered to produce sharpness as a function of viewer distance, more accurately simulating a camera with a finite depth of field.

Depth-of-field and stereo viewing are similar. In both cases, there is more than one viewpoint, with all view directions converging at a fixed distance along the direction of view. When computing depth of field tranforms, however, we only use shear instead of rotation, and sample a number of viewpoints, not just two, along an axis perpendicular to the view direction. The resulting images are blended together.

This process creates images whose the objects in front of and behind the fusion distance shift position as a function of viewpoint. In the blended image, these objects appear blurry. The closer the object is to the fusion distance, the less it shifts, and the sharper they appear.

The field of view can be expanded by increasing the ratio between the viewpoint shift and fusion distance. This way objects have to be farther from the fusion distance to shift significantly.

For details on rendering scenes featuring a limited field of view see Section 9.1.

4.3 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 `glDepthRange` mapping specifies a transformation for the z coordinate similar to the viewport transformation used to map x and y to window coordinates. The `glDepthRange` 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 `glDepthRange` 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 `glDepthRange` call. The mapping may 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 `glDepthRange(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.

To understand why there might be this disparity in the rendering quality, its important to understand the characteristics of the window coordinate z coordinate. The z value does specify the distance from the fragment to the plane of 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 amount 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 9 plots the window coordinate z value as a function of the eye-to-pixel distance for several ratios of far to near. The non-linearity increases the resolution of the z-values when they are close to the near clipping plane, increasing the resolving power 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 doing this is that objects rendered 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 window 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 window 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. This can cause noticeable Mach bands in fogged scenes.

4.3.1 Depth Buffering

We have discussed some of the caveats of using depth buffering, but there are several other aspects of OpenGL rasterization and depth buffering that are worth mentioning [2]. One big problem is that the rasterization process uses inexact arithmetic so it is exceedingly difficult to handle primitives that are coplanar unless they

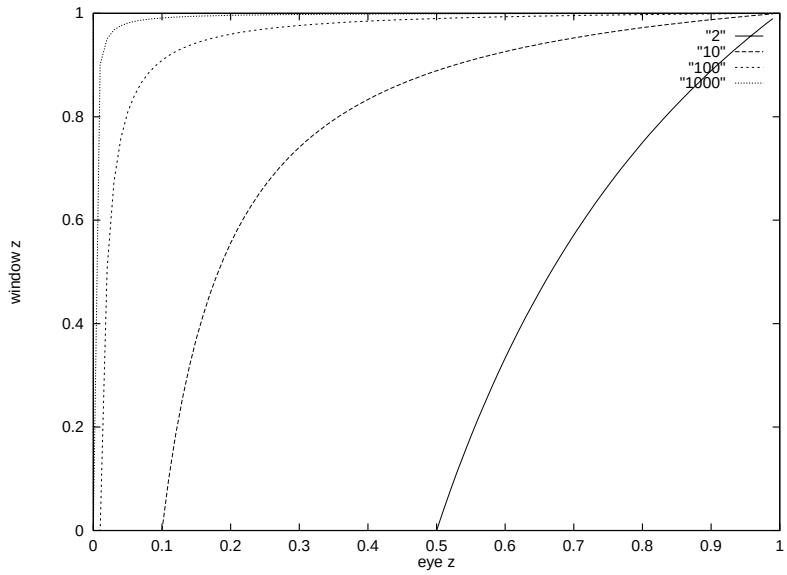


Figure 9. The relationship of window z (depth) to eye z for different far/near ratios

share the same plane equation. This problem is exacerbated by the finite precision of depth buffer implementations. Many solutions have been proposed to handle this class of problems, which involve coplanar primitives:

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 alternative methods to get more insight into the uses of the depth buffer.

The problem of decaling one coplanar polygon into another 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 within 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 removed by outlining polygons with one color and drawing the filled polygons with the background color. Ideally a polygon could be outlined by simply connecting the vertices together with line primitives. This seems 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 fully contained within the base polygon.

The solution most frequently suggested for this problem is to draw the outline as a series of lines and translate the outline a small amount towards the eye. Alternately, the polygon could be translated away from the eye instead. Besides not

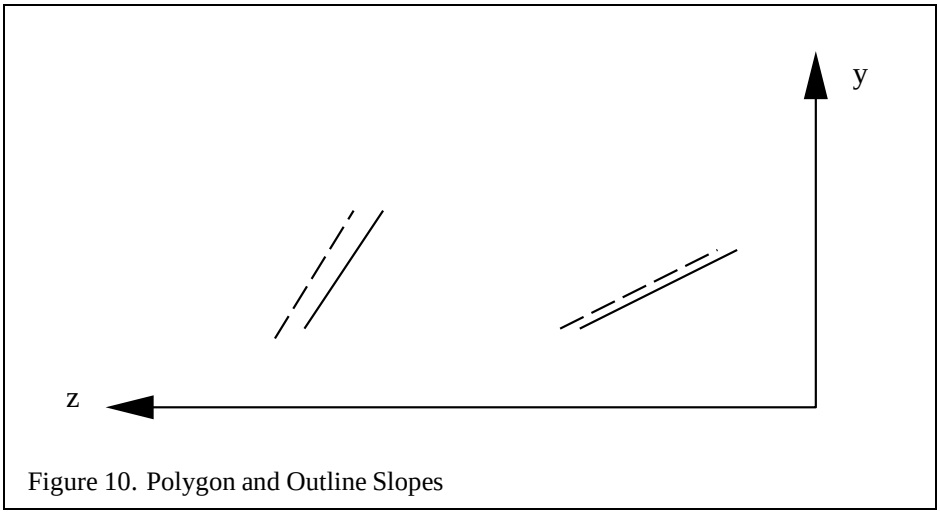


Figure 10. Polygon and Outline Slopes

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 10 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 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. The figure shows 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 the `EXT_polygon_offset` extension that provides a scaled slope plus bias capability for solving outline problems such as these and for other applications. A modified version of this polygon offset extension has been added to the core of OpenGL 1.1 as well.

4.4 Image Tiling

When rendering a scene in OpenGL, the resolution of the image is normally limited to the workstation screen size. For interactive applications this is usually sufficient, but there may be times when a higher resolution image is needed. Examples include color printing applications and computer graphics recorded for film. In these cases, higher resolution images can be divided into tiles that fit on the workstation's frame buffer. The image is rendered tile by tile, with the results saved into off screen memory, or perhaps a file. The image can then be sent to a printer or film recorder, or undergo further processing, such has downsampling to produce an antialiased image.

One very straightforward way to tile an image is to manipulate the `glFrustum` call's arguments. The scene can be rendered repeatedly, one tile at a time, by changing the left, right, bottom and top arguments arguments of `glFrustum` for each tile.

Computing the argument values is straightforward. Divide the original width and height range by the number of tiles horizontally and vertically, and use those values to parametrically find the left, right, top, and bottom values for each tile.

$$tile(i,j); i:0 \rightarrow nTiles_{horiz}, j:0 \rightarrow nTiles_{vert}$$

$$right_{tiled}(i) = left_{orig} + \frac{right_{orig} - left_{orig}}{nTiles_{horiz}} * (i+1)$$

$$left_{tiled}(i) = left_{orig} + \frac{right_{orig} - left_{orig}}{nTiles_{horiz}} * i$$

$$top_{tiled}(j) = bottom_{orig} + \frac{top_{orig} - bottom_{orig}}{nTiles_{vert}} * (j+1)$$

$$bottom_{tiled}(j) = bottom_{orig} + \frac{top_{orig} - bottom_{orig}}{nTiles_{vert}} * j$$

In the equations above, each value of i and j corresponds to a tile in the scene. If the original scene is divided into $nTiles_{horiz}$ by $nTiles_{vert}$ tiles, then iterating through the combinations of i and j generate the left, right top, and bottom values for `glFrustum` to create the tile.

Since `glglFrustum` has a shearing component in the matrix, the tiles stitch together seamlessly to form the scene. Unfortunately, this technique would have to be modified for use with `gluPerspective` or `glOrtho`. There is a better approach, however. Instead of modifying the perspective transform call directly, apply tranforms to the results. The area of normalized device coordinate (NDC) space corresponding to the tile of interest is translated and scaled so it fills the NDC cube.

Working in NDC space instead of eye space makes finding the tiling tranforms easier, and is independent of the type of projective transform.

Even though it's easy to visualize the operations happening in NDC space, conceptually, you can "push" the transforms back into eye space, and the technique maps into the `glFrustum` approach described above.

Tor the transform operations to happen after the projection transform, the OpenGL calls must happen before it. Here is the sequence of operattions:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glScalef(xScale, yScale);
glTranslatef(xOffset, yOffset, 0.f);
setProjection();
```

The scale factors xScale and yScale scale the tile of interest to fill the the entire scene:

$$xScale = \frac{sceneWidth}{tileWidth}$$
$$yScale = \frac{sceneHeight}{tileHeight}$$

The offsets xOffset and yOffset are used to offset the tile so it is centered about the Z axis. In this example, the tiles are specified by their lower left corner relative to their position in the scene, but the translation needs to move the center of the tile into the origin of the X-Y plane in NDC space:

$$xOffset = \frac{-2 * left}{sceneWidth} + (1 - \frac{1}{nTiles_{horiz}})$$
$$yOffset = \frac{-2 * bottom}{sceneHeight} + (1 - \frac{1}{nTiles_{vert}})$$

As before $nTiles_{horiz}$ is the number of tiles that span the scene horizontally, while $nTiles_{horiz}$ is the number of tiles that span the scene vertically.

Some care should be taken when computing $left$, $bottom$, $tileWidth$ and $tileHeight$ values. It's important that each tile is abutted properly with it's neighbors. Ensure this by guarding against round-off errors. Some code that properly computes these values is given below:

```
/* tileWidth and tileHeight are GLfloats */
GLint bottom, top;
GLint left, right;
GLint width, height;
for(j = 0; j < num_vertical_tiles; j++) {
        for(i = 0; i < num_horizontal_tiles; i++) {
                left = i * tileWidth;
                right = (i + 1) * tileWidth;
                bottom = j * tileHeight;
                top = (j + 1) * tileHeight;
                width = right - left;
                height = top - bottom;
                /* compute xScale, yScale, xOffset, yOffset */
        }
}
```

Note that the parameter values are computed so that $left + tileWidth$ is guaranteed to be equal to $right$ and equal to $left$ of the next tile over, even if $tileWidth$ has a fractional component? If the frustum technique is used, similar precautions should be taken with the $left$, $right$, $bottom$, and top parameters to `glFrustum`.

4.5 Moving the Current Raster Position

Using the `glRasterPos` command, the raster position will be invalid if the specified position was culled. Since `glDrawPixels` and `glCopyPixels` operations applied when the raster position is invalid do not draw anything, it may seem that the lower left corner of a pixel rectangle must be inside the clip rectangle. This problem may be overcome by using the `glBitmap` command. The `glBitmap` command takes arguments *xoff* and *yoff* which specify an increment to be added to the current raster position. Assuming the raster position is valid, it may be moved outside the clipping rectangle by a `glBitmap` command. `glBitmap` is often used with a 0 size rectangle to move the raster position.

5 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 that 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 1.1 readily supports the first two techniques (surface color manipulation and environment mapping). Texture mapping can also solve some rendering problems in less obvious ways. This section reviews some of the details of OpenGL texturing support, outline some considerations when using texturing and suggest some interesting algorithms using texturing.

5.1 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 [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 resulting in a texture *tile* that repeats across the surface. 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.

5.1.1 Filtering

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 choose some filter which does interpolation. For magnification, OpenGL 1.1 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 1.1 supports various types of mipmapping [51], 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.

5.1.2 Texture Environment

The process by which the final fragment color value is derived is called the texture environment function (`glTexEnv`) Several methods exist for computing the final color, each capable of producing a particular effect. One of the most commonly used is the modulate function. For all practical purposes the modulate function multiplies or modulates 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 that 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. The blend environment function uses the texture value to control the mix of the incoming fragment color and a constant texture environment color. OpenGL 1.1 adds a replace texture environment which substitutes the texel color for the incoming fragment color. This effect can be achieved using the modulate environment, but replace has a lower computational burden.

Another useful (and sometimes misunderstood) feature of OpenGL is the texture border. OpenGL supports either a constant texture border color or a border that is 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 `GL_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), however, the resulting texel value is a 50% mix of the border color and the outer texel of the texture image at that edge.

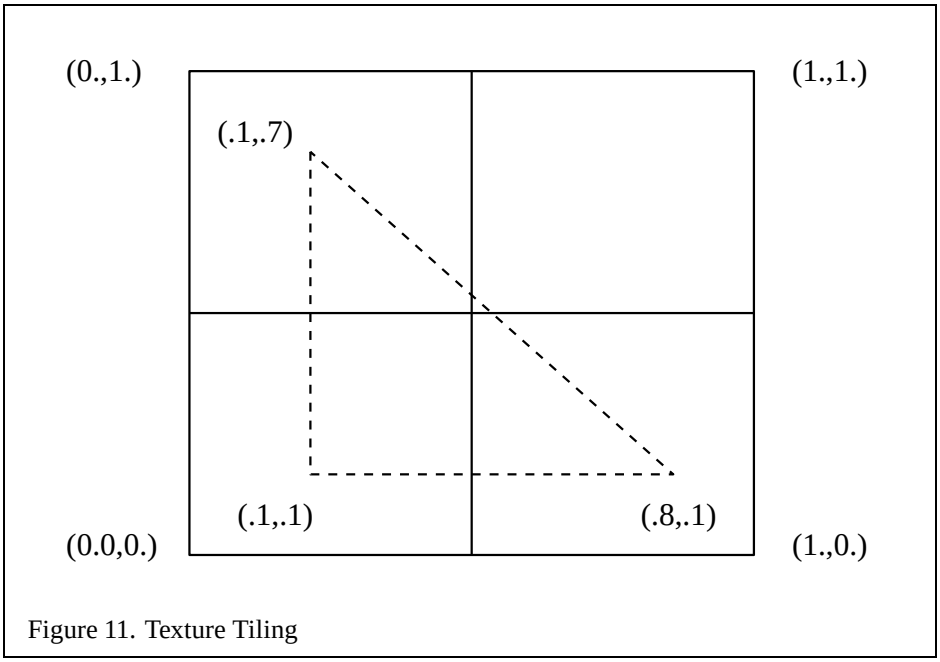


Figure 11. Texture Tiling

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 11. 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.

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 that are proportional to the texture coordinate values and use `glAlphaFunc` 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 map to texels with an alpha of 1.0 for pixels to be retained and 0.0 for pixels to be rejected.

5.2 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 highest quality of the minification algorithms being `GL_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 repeatedly 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 [30] 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 filtering algorithm other than the one supplied by the GLU library, it is important to keep a couple of things in mind. The highest res-

olution 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 not true for other filter functions. 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, filters with widths wider than 2 need to sample outside the boundaries of the image. This is commonly handled 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 `GL_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 quad 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. The `glCopyTexImage(c)`apability, added in OpenGL 1.1, allows an image in the color buffer to be copied directly to texture memory.

This process can still be slightly difficult in OpenGL 1.0 as it 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 in OpenGL 1.1 with texture objects which allow multiple texture definitions to coexist at the same time. However, it would be much simpler if we could use the most recent level loaded as part of the mipmap as the current texture for drawing. OpenGL 1.1 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 (GL_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 GL_REPEAT wrap mode can be used.

5.3 View Dependent Filtering

OpenGL specifies an isotropic filter for texture minification. This means that the amount of filtering done along the s and t axes of the texture is the same, and is the maximum of the filtering needed along each of the two axes individually. This can lead to excessive blurring when a texture is viewed at an angle. If it is known that a texture will always be viewed at a given angle or range of angles, it can be created in a way that reduces overfiltering.

Suppose a textured square is rendered as shown in the left of Figure 12. The texture is shown in the right. Consider the fragment that is shaded dark. Its ideal footprint is shown in the diagram of the texture as the dark inner region. But since the minification filter is isotropic, the actual footprint is forced to a square that encloses the dark region. A mipmap level will be chosen in which this square footprint is properly filtered for the fragment; in other words, a mipmap level will be selected in which the size of this square is closest to the size of the fragment. That mipmap is not level zero but level 1 or higher. Hence, at that fragment more filtering is needed along t than along s, but the same amount of filtering is done in both.

To avoid this problem, we do the extra filtering along t ourselves when we cre-

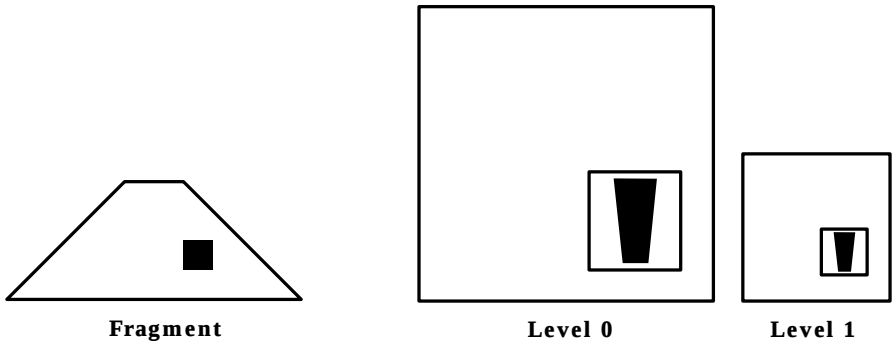


Figure 12. Footprint in full height texture

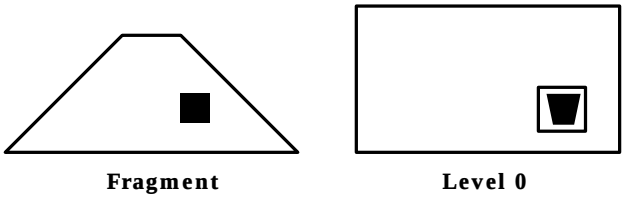


Figure 13. Footprint in half height texture

ate the texture, and make the texture have the same width but only half the height. See Figure 13. The footprint now has an aspect ratio that is more square, so the enclosing square is not much larger, and is closer to the size to the fragment. Level 0 will be used instead of a higher level. Another way to think about this is that by using a texture that is shorter along t, we reduce the amount of minification that is required along t.

5.4 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 it is 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.

5.5 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 [41].

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 `GL_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 `glTexSubImage` routine allows a subregion within an existing texture image to be updated. This makes 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 14, 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.

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 a 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 the 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.

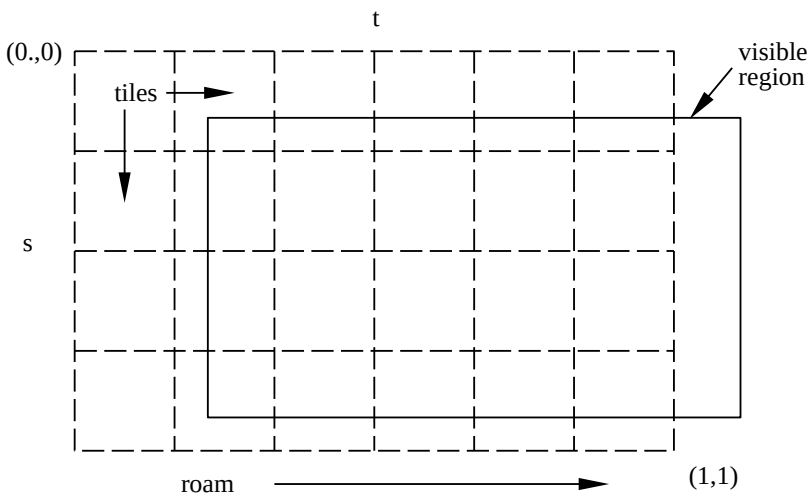


Figure 14. 2D Image Roam

5.6 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 GL_EQUALS test. The advantage of using the alpha test over blending is that blending typically 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 antialiased.

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.

5.7 Billboards

It is often desirable to replace intricate geometry with simpler texture mapped geometry to increase realism and performance. Billboarding is a technique in which complex objects such as trees are drawn with simple planar texture mapped geometry and the geometry is transformed to face the viewer. The transformation typically consists of a rotation to orient the object towards the viewer and a translation to place the object in the correct position . For the case of the tree, an object with roughly cylindrical symmetry, an axial rotation is used to rotate the geometry for the tree, typically a quadrilateral, about the axis running parallel to the tree trunk.

For the simple case of the viewer looking down the negative z-axis and the up vector equal to the positive y-axis, the angle of rotation can be determined by computing the eye vector from the model view matrix M

$$\vec{V}_{eye} = M^{-1} \begin{pmatrix} 0 \\ 0 \\ -1 \\ 0 \end{pmatrix}$$

and the rotation θ about the y axis is computed as

$$\begin{aligned} \cos\theta &= \vec{V}_{eye} \cdot \vec{V}_{front} \\ \sin\theta &= \vec{V}_{eye} \cdot \vec{V}_{right} \end{aligned}$$

where

$$\begin{aligned} \vec{V}_{front} &= (0, 0, 1) \\ \vec{V}_{right} &= (1, 0, 0) \end{aligned}$$

Once θ has been computed a rotation matrix R can be constructed for the rotation about the y-axis ($\vec{V}_{up}$) and combined with the model view matrix as MR and used to transform the billboard geometry.

To handle the more general case of an arbitrary billboard rotation axis, an intermediate alignment rotation A of the billboard axis into the $\vec{V}_{up}$ axis is computed as

$$\begin{aligned} \vec{axis} &= \vec{V}_{up} \times \vec{V}_{billboard} \\ \cos\theta &= \vec{V}_{up} \cdot \vec{V}_{billboard} \\ \sin\theta &= \|\vec{axis}\| \end{aligned}$$

and the matrix transformation is replaced with MAR. Note that the preceding calculations assume that the projection matrix contains no rotational component.

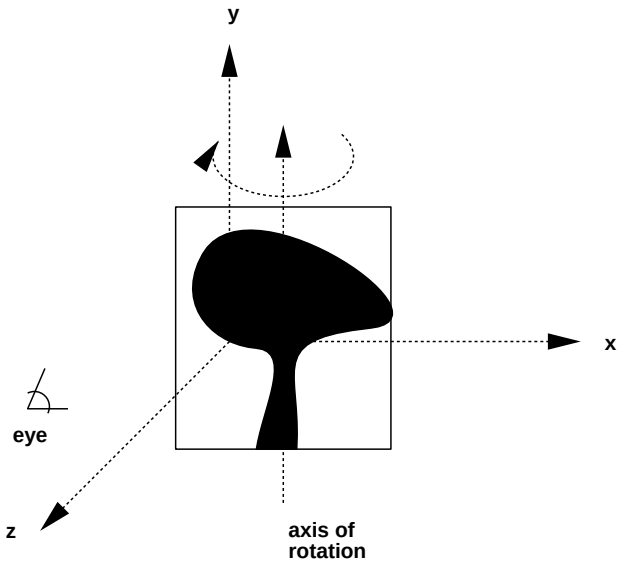


Figure 15. Billboard with cylindrical symmetry

In addition to objects which are cylindrically symmetric, it is also useful to compute transformations for spherically symmetric objects such as smoke, clouds and bushes. Spherical symmetry allows billboards to rotate up and down as well as left and right, whereas cylindrical behavior only allows rotation to the left or right. Cylindrical behavior is suited to objects such as trees which should not bend backward as the viewer's altitude increases.

Objects which are spherically symmetric are rotated about a point to face the view and thus provide more freedom in computing the rotations. An additional alignment constraint can be used to resolve this freedom. For example, an alignment constraint which keeps the object oriented in a consistent fashion, such as upright. This constraint can be enforced in object coordinates when the objective is to maintain scene realism, perhaps to maintain the orientation of plume of smoke consistently with other objects in a scene. The constraint can also be enforced in eye coordinates which can be used to maintain alignment of an object relative to the screen, for example, keeping annotations such as text aligned horizontally on the screen.

The computations for the spherically symmetric case are a minor extension of the computations for the arbitrarily aligned cylindrical case. First an alignment transformation, A, is computed to rotate the alignment axis onto the up vector followed by a rotation about the up vector to align the face of the billboard with the eye vector. A is computed as

$$\begin{aligned} \vec{axis} &= \vec{V}_{up} \times \vec{V}_{alignment} \\ \cos\theta &= \vec{V}_{up} \cdot \vec{V}_{alignment} \\ \sin\theta &= \|\vec{axis}\| \end{aligned}$$

where $\vec{V}_{alignment}$ is the billboard alignment axis with the component in the direction of the eye direction vector removed

$$\vec{V}_{alignment} = \vec{V}_{billboard} - (\vec{V}_{eye} \cdot \vec{V}_{billboard})\vec{V}_{eye}$$

A rotation about the up vector is then computed as for the cylindrical case.

5.8 Rendering Text

A novel use for texturing is rendering antialiased text [20]. 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 positioning 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 filtering the images to produce the levels of detail.

5.9 Projective Textures

Projective textures [44] 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.

5.10 Environment Mapping

OpenGL directly supports environment mapping using spherical environment maps. A sphere map is a single texture of a perfectly reflecting sphere in the envi-

ronment 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 (GL_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 [45]. We discuss this algorithm in a later section.

5.11 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.

5.12 3D Textures

Three dimensional textures are a logical extension of 2D textures. In 3D textures, texels become unit cubes in texel space. They are packed into a rectangular parallelpiped, each dimension contrained to be a power of two. This texture map occupies a volume, rather than a rectangular region, and is accessed using three texture coordinates, S, T, and R. As with 2D textures, texture coordinates range from zero to 1 in each dimension. Filtering is controlled in the same fashion as 2D textures, using texture parameters and texture environment.

5.12.1 Using 3D Textures

In OpenGL, 3D textures have much in common with 2D and 1D textures. Texture parameters and texture environment calls are the same, using the GL_TEXTURE_3D_EXT target in place of GL_TEXTURE_2D or GL_TEXTURE_1D.

Internal and External Formats and Types are the same, although a particular OpenGL implementation may limit the 3D texture formats.

3D textures need to be accessed with S, T, and R texture coordinates instead of just S and T. The additional texture coordinate complexity, combined with the

common uses for 3D textures, means texture coordinate generation is used more commonly for 3D textures than for 2D and 1D.

3D texture maps take up a large amount of texture memory, and are expensive to change dynamically. This can affect multipass algorithms that require multiple passes with different textures.

The texture matrix operates on 3D texture coordinates in the same way that it does for 2D and 1D textures. A 3D texture volume can be translated, rotated, scaled, or have other transforms applied to it. Applying a transformation to the texture matrix is a convenient and high performance way to manipulate a 3D texture when it is too expensive to alter the texel values directly.

3D Textures vs. MIPmaps A clear distinction should be made between 3D textures and MIPmapped 2D textures. 3D textures can be thought of as a solid block of texture, requiring a third texture coordinate R, to access any given texel. A 2D MIPmap is a series of 2D texture maps, each filtered to a different resolution. Texels from the appropriate level(s) are chosen and filtered, based on the relationship between texel and pixel size on the primitive being textured.

Like 2D textures, 3D texture maps can be MIPmapped. Instead of resampling a 2D layer, the entire texture volume is filtered down to an eighth of its volume by averaging eight adjacent texels on one level down to a single texel on the next. MIPMapping serves the same purpose in both 2D and 3D texture maps; it provides a means of accurately filtering when the projected texel size is small relative to the pixels being rendered.

3D texture mipmapping is not widely supported, mostly because it is unnecessary for the most common use of 3D textures, volume visualization. Nevertheless, some systems support it, and it can be used for rendering solids as discussed below.

5.12.2 3D Texture Portability

3D Textures aren't currently a core feature in OpenGL, but can be accessed as an extension. It is an *EXT* extension, indicating more than one vendor supports it. Even when 3D texture maps are supported, the application writer must be careful to consider the level of support present in the application. Texture map size may be limited, and 3D MIPmapping is often not supported. Available internal and external formats and types may be restricted. All of these restrictions can be queried at run time, and with care, portable code can be produced.

Consider writing your 3D texture applications so that they revert to a 2D texturing mode if 3D textures aren't supported. See the volume visualization section for an example of a 3D texture algorithm that will work, with lower quality, using 2D textures.

5.12.3 3D Textures to Render Solid Materials

A direct 3D texture application is rendering solid objects composed of heterogeneous material. An example is rendering a statue made of marble or wood. The object itself is composed of polygons or NURBS surfaces bounding the solid. Combined with proper texgen values, rendering the surface using a 3D texture of the material makes the object appear cut out of the material. With 2D textures objects often appear to have the material laminated on the surface. The difference can be striking when there are obvious 3D coherencies in the material, combined with sharp angles in the object's surface.

Rendering a solid with 3D texture is straightforward:

Create the 3D texture The texture data for the material is organized as a three dimensional array. Often the material is generated procedurally. As with 2D textures, proper filtering and sampling of the data must be done to avoid aliasing. A MIPmapped 3D texture will increase realism of the object. OpenGL doesn't support a `gluBuild3DMipmap` command, so the mipmaps need to created by the application. Be sure to check to see if the size of the texture you want to create is supported by the system, and there is sufficient texture memory available by calling `glTexImage3DEXT` with `GL_PROXY_TEXTURE_3D_EXT` to find a supported size. You can also call `glGet` with `GL_MAX_3D_TEXTURE_SIZE_EXT` to find the maximum allowed size of any dimension in a 3D texture for your implementation of OpenGL, though the result may be more conservative than the result of a `PROXY` query.

Create Texture Coordinates For a solid surface, using `glTexGen` to create the texture coordinates is the easiest approach. Define planes for S, T, and R in eye space. Adjusting the scale has more effect on texture quality than the position and orientation of the planes, since scaling affects how the texture is sampled.

Enable Texturing Use `glEnable(GL_TEXTURE_3D_EXT)` to enable 3D texture mapping. Be sure to set the texture parameters and texture environment appropriately. Check to see what restrictions your implementation puts on these values.

Render the Object Once configured, rendering with 3D texture is no different than other texturing.

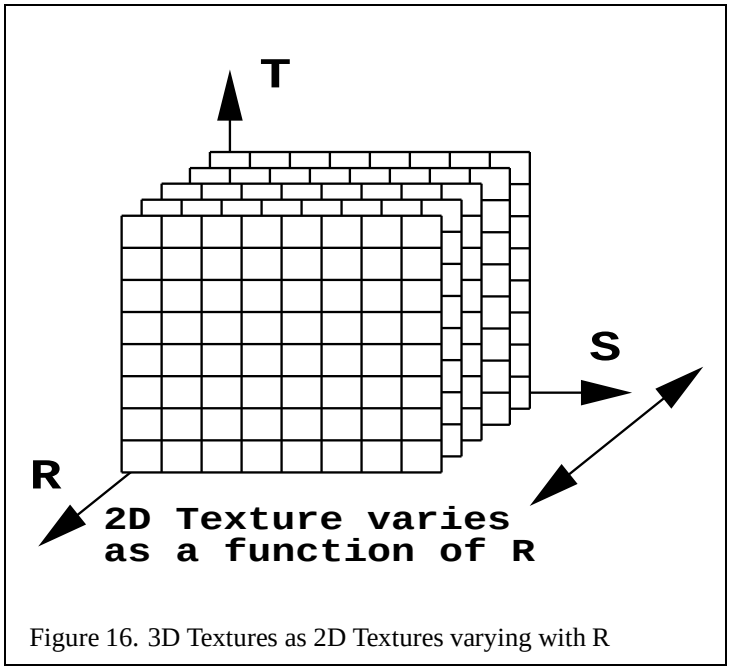


Figure 16. 3D Textures as 2D Textures varying with R

5.12.4 3D Textures as Multidimensional Functions

Instead of thinking of a 3D texture as a 3D volume of data, it can be thought of as a 2D texture map that varies as a function of the R coordinate value. Since the 3D texture filters in three dimensions, changing the R value smoothly blends from one 2D texture image to the next.

An obvious application is animated 2D textures. A 3D texture can animate a sequence of images by using the R value as time. Since the images are interpolated, temporal aliasing is reduced.

Another application is generalized billboards. A normal billboard is a 2D texture applied to a polygon that always faces the viewer. Billboards of objects such as trees behave poorly when the viewer views the object from above. A 3D texture billboard can change the textured image as a function of viewer elevation angle, blending a sequence of images between side view and top view, depending on the viewer's position.

5.13 Procedural Texture Generation

Procedurally generated textures are a diverse topic; we concentrate on those based on *filtered noise functions* . They are commonly used to simulate effects from phe-

nomena such as fire, smoke, clouds, and marble formation. These textures are described in detail in [13], which provides the basis for much of this section.

5.13.1 Filtered Noise Functions

A filtered noise function is simply a function created by filtering impulses of random amplitude over the domain. There are a variety of ways to distribute the impulses spatially and to filter those impulses; these methods determine the character of the function and, in turn, the character of the procedural texture created from the function. Regardless of the method chosen, a filtered noise function should have certain properties [13], some of which are:

- It is a repeatable pseudorandom function of its inputs.
- It has a known range, typically -1 to 1.
- It is band-limited, with a maximum frequency of about 1 per domain unit.

Given such a function, we can build a more interesting function by making dilated versions of the original such that each one has a frequency of 2, 4, 8, etc. These are called the *octaves* of the original function. The octaves are then composited together with the original noise function using some set of weights. The result is a band-limited function which gives the impression of controlled randomness in each frequency band.

One way of distributing noise impulses is to space them uniformly along the coordinate axes, as in a lattice. In *value noise* , the function itself interpolates the values at the lattice points, while in *gradient noise* the gradient of the function interpolates the values at the lattice points [13]. Gradient noise is similar to the noise function implemented in the RenderMan shading language.

Lattice noises can exhibit axis-aligned artifacts. Lewis [29] describes *sparse convolution* , a way to avoid such artifacts by distributing the impulses using a stochastic process, and van Wijk [47] describes a similar technique called *spot noise*.

Although the noise functions described in [13] are generally 3D, we first discuss how to generate a 2D noise function, because it is more straightforward to construct in a 2D framebuffer and because some simple interesting effects can be created with it.

5.13.2 Generating Noise Functions

Filtered noise functions are typically implemented as continuous functions that can be sampled at an arbitrary domain value. However, for some applications a set of

uniformly spaced samples of the function may suffice. In these cases, a discrete version of the function can be created in the framebuffer using OpenGL. In the following, we do not distinguish between the terms *noise function* and *discrete noise function* .

A simple way to create lattice noise is to create a texture with random values for the texels, and then to draw a textured rectangle with a bilinear texture filter at an appropriate magnification. However, bilinear interpolation produces poor results, especially when creating the lower octaves, where values are interpolated across a large area. Some OpenGL implementations support bicubic texture filtering, which may produce results of acceptable quality. However, a particular implementation of bicubic filtering may have limited subtexel precision, causing noticeable banding at the lower octaves. Both bilinear and bicubic filters also have the limitation that they produce only value noise; gradient noise is not possible. We suggest another approach.

5.13.3 High Resolution Filtering

The accumulation buffer can be used to convolve a high resolution filter with a relatively small image under magnification. That is what we need to make the different octaves; the octave representing the lowest frequency band will be created from a very small input image under large magnification. Suppose we want to create a 512x512 output image by convolving a 64x64 filter with a 4x4 input image. Our filter takes a 2x2 array of samples from the input image at a time, but is discretized into 64x64 values in order to generate an output image of the desired size. The input image is shown on the left in Figure 17 with each texel numbered. The output image is shown on the left in Figure 18. Note that each texel of the input image will make a contribution to a 64x64 region of the output image. Consider these regions for texels 5, 7, 13, and 15 of the input image; they are adjacent to each other and have no overlap, as shown by the dotted lines on the left in Figure 18. Hence, these four texels can be evaluated in the same pass without interfering with each other. Making use of this fact, we redistribute the texels of the input image into four 2x2 textures as shown in the right of Figure 17. We also create a 64x64 texture that contains the filter function; this texture will be used to modulate the contribution of the input texel over a 64x64 region of the color buffer. The steps to evaluate the texels in Texture D are:

1. Using the filter texture, draw four filter functions into the alpha planes with the appropriate x and y offset, as shown on the right in Figure 18

2. Enable alpha blending and set the source blend factor to GL_DST_ALPHA and the destination blend factor to GL_ZERO.

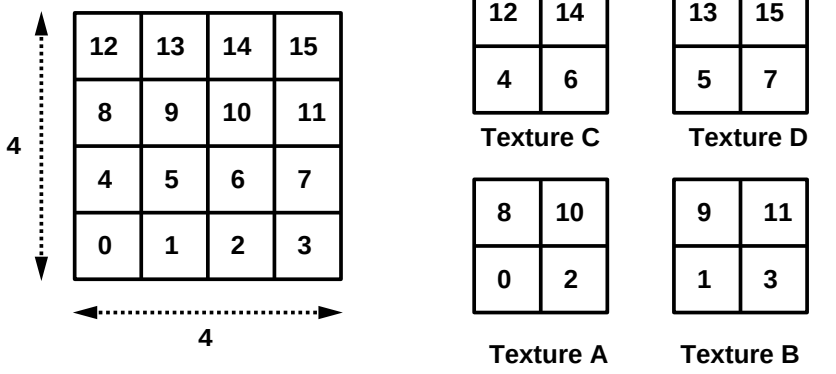


Figure 17. Input Image

3. Set the texture magnification filter to GL_NEAREST.

4. Draw a rectangle to the dotted region with Texture D, noting the offset of 64 pixels in both x and y.

5. Accumulate the result into the accumulation buffer.

Repeat the above procedure for Textures A, B, and C with the appropriate x and y offsets, and return the contents of the accumulation buffer to the color buffer.

A wider filter requires more passes of the above procedure, and also requires that the original texture be divided into more small textures. For example, if we had chosen a filter that covers a 4x4 array of input samples instead of 2x2, we would have to make 16 passes instead of 4, and we would have to distribute the texels into 16 1x1 textures. Increasing the size of either the output image or the input image, however, has no effect on the number of passes.

5.13.4 Spectral Synthesis

Now that we can create a single frequency noise function using the framebuffer, we need to create the different octaves and to composite them into one texture. For each octave:

1. Scale the texture matrix by a power of 2 in both s and t.

2. Translate the texture matrix by a random offset in both s and t.

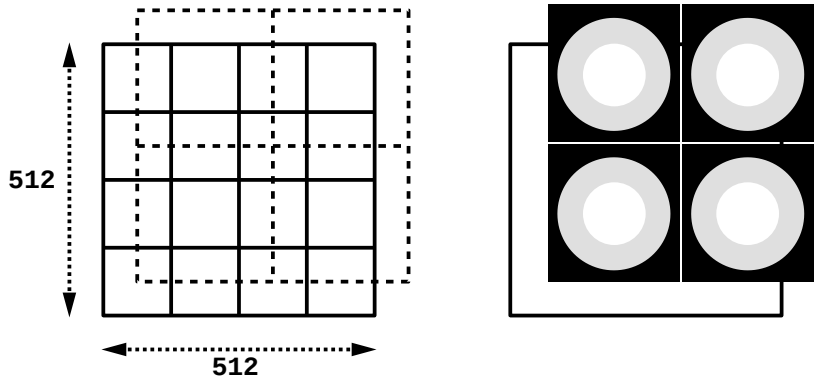


Figure 18. Output Image

3. Set the texture wrap mode to GL_REPEAT for s and t.
4. Draw a textured rectangle.
5. Accumulate the color buffer contents.

The random translation is an attempt to minimize the amount of overlap between each octave's texels; without it, every octave would use texels from the same corner of the input image. The accumulation is typically done with a scale factor that controls the weight we want to give each octave.

5.13.5 Other Noise Functions

Gradient noise can be created using the same method described above, but with a different filter. The technique described above can also create noise that is not aligned on a lattice. To create sparse convolution noise [29] or spot noise [47], instead of drawing the entire point-sampled texture at once, draw one texel and one copy of the filter at a time for each random location.

5.13.6 Turbulence

To create an illusion of turbulent flow, first-derivative discontinuities are introduced into the noise function by taking the absolute value of the function. Although OpenGL does not include an absolute value operator for framebuffer contents, the same effect can be achieved by the following:

1. `glAccum(GL_LOAD,1.0);`
2. `glAccum(GL_ADD,-0.5);`
3. `glAccum(GL_MULT,2.0);`
4. `glAccum(GL_RETURN,1.0);`
5. Save the image in the color buffer to a texture, main memory, or other color buffer.
6. `glAccum(GL_RETURN,-1.0);`
7. Draw the saved image from Step 4 using `GL_ONE` as both the source blend factor and the destination blend factor.

The calls with `GL_ADD` and `GL_MULT` map the values in the accumulation buffer from the range [0,1] to [-1,1]; this is needed because values retrieved from the color buffer into the accumulation buffer are positive. Since values from the accumulation buffer are clamped to [0,1] when returned, the first `GL_RETURN` clamps all negative values to 0 and returns the positive values intact. The second `GL_RETURN` clamps the positive values to 0, and negates and returns the negative values. The color buffer needs to be saved after the first `GL_RETURN` because the second `GL_RETURN` overwrites the color buffer; OpenGL does not define blending for accumulation buffer operations.

5.13.7 Example: Image Warping

A common use of a 2D noise texture is to distort the texture coordinates while drawing a 2D image, thus warping the image. A noise function is created in the framebuffer as described above, read back to the host, and used as texture coordinates (or offsets to texture coordinates) to render the image. Since color values in OpenGL are normalized to the range 0.0 to 1.0, if one is careful the image returned to the host may be used without much conversion; assuming that the modelview and texture matrixes are set up to accept values in this range, the returned data may be used directly for rendering.

Another similar use of a 2D noise texture is to distort the reflection of an image. In OpenGL, reflections on a flat surface can be done by reflecting a scene across the surface. The results can be copied from the framebuffer to texture memory, and in turn drawn with distorted texture coordinates. The shape and form of the distortion can be controlled by modulating the contents of the framebuffer after the noise texture is drawn but before it is copied to texture memory. This can produce interesting effects such as water ripples.

5.13.8 Generating 3D Noise

Using the techniques described above for generating a 2D noise function, we can generating a 3D noise function by making 2D slices and filtering them. A 2D slice spans the s and t axes of the lattice, and corresponds to a slice of the lattice at a fixed r.

Suppose we want to make a 64x64x64 noise function with a frequency of 1 per domain unit, using the same filtering (but one that now takes 2x2x2 input samples) as in the 2D example above. We first create 2 slices, one for r=0.0 and one for r=1.0. Then we create the 62 slices in between 0 and 1 by interpolating the two slices. This interpolation can take place in the color buffer using blending, or it can take place in the accumulation buffer. Functions with higher frequencies are created in a similar way. Widening the filter dramatically increases the number of passes; going from a 2x2x2 filter to 4x4x4 requires 16 times as many passes.

To synthesize a function with different frequencies, we create a 3D noise function for each frequency, and composite the different frequencies using a set of weights, just as we do in the 2D case. It is clear that a large amount of memory is required to store the different 3D noise functions. These operations may be reordered so that less total memory is required, perhaps at the expense of more interpolation passes.

5.13.9 Generating 2D Noise to Simulate 3D Noise

We have described a method for creating 2D noise functions. In the case of lattice noise, these 2D functions correspond to a 2D slice of the lattice. There are cases where we want to model a 3D noise function and where such a 2D function is inadequate. For example, to draw a vase that looks like it was carved from a solid block of marble, we cannot use a lattice 2D noise function.

However, we can create a 2D noise function that approximates the appearance of a true 3D noise function, using spot noise [47]. We take into account the object space coordinates of the geometry, and generate only spots that are close enough to the geometry to make a contribution to the 3D noise at those points. The difficulty is how to render the spot in such a way that at each fragment the value of the spot is determined by the object space distance from the center of the spot to that fragment. Depending on the complexity of the geometry, we may be able to make an acceptable approximation to the correct spot value by distorting the spot texture. One possible way to improve the approximation is to compensate for a nonuniform mapping of the noise texture to the geometry. Van Wijk describes how he does this by nonuniformly scaling a spot. Approximating the correct spot value is most important when generating the lower octaves, where the spots are largest and errors

are most noticeable.

5.13.10 Trade-offs Between 3D and 2D Techniques

A 3D texture can be used with arbitrary geometry without much additional work if your OpenGL implementation supports 3D textures. However, generating a 3D noise texture requires a large amount of memory and a large number of passes, especially if you use a filter that convolves a large number of input values at a time. A 2D texture as we just described doesn't require nearly as many passes to create, but it does require knowledge of the geometry and additional computation in order to properly shape the spot.

6 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 `GL_ZERO`, `GL_ONE`, `GL_SRC_ALPHA` and `GL_ONE_MINUS_SRC_ALPHA`. OpenGL 1.1 specifies additive blending, but 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.

6.1 Compositing

The OpenGL blending operation does not directly implement the compositing operations described by Porter and Duff [38]. 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, it's 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 `glCopyPixels` operation with blending enabled and the blending function set to (`GL_SRC_ALPHA`,`GL_ZERO`). To ensure that the original alpha value is left intact, use the `glColorMask` command to disable updates to the alpha component during the copy operation.

6.2 Advanced Blending

OpenGL 1.1 blending only allows simple additive combinations of the source and destination color components. Two ways in which the blending operations have been extended by 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 1.1, 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).

6.3 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 or 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.

6.4 Blending with 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 color buffer.

The accumulation buffer can be cleared like any other buffer. You can use `glClearAccum` to set the red, green, blue, and alpha components of its clear color. 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 color buffer, then use `glAccum` to accumulate that image into the accumulation buffer. The `glAccum` command reads from the currently selected read buffer. 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 possible settings for op are described in Table reftab:accumop.

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 of the accumulation buffer are copied 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.

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.

Table 1: `glAccum` op values

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 settings are the default.

3. Clear the back buffer with `glClear`, 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 [21, 33]. They include 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.

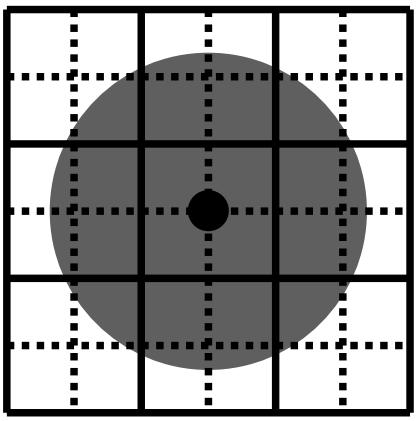


Figure 19. Rasterization of a wide point.

7 Antialiasing

Aliasing artifacts appear when rasterizing primitives because primitives are approximated by a series of pixels that lie on an integer grid. Aliasing is especially bad when rendering diagonal lines (or edges of polygons that are on the diagonal) and wide points. Figure 19 shows how a wide point covers more of some pixel squares than others.

Antialiasing is a technique that reduces aliasing artifacts (or jaggies) by modifying the intensity of a primitive's fragment based on how much the primitive overlaps that pixel fragment. When performing antialiasing, OpenGL calculates a coverage value for each pixel fragment based on the how much the primitive overlaps that pixel.

7.1 Antialiasing Points and Lines

To antialias points or lines, you need to enable antialiasing by calling `glEnable` and passing in `GL_POINT_SMOOTH` or `GL_LINE_SMOOTH`, as appropriate. You can also provide a quality hint by calling `glHint`. The hint parameter can be `GL_FASTEST` to indicate that the most efficient option should be chosen, `GL_NICEST` to indicate the highest quality option should be chosen, or `GL_DONT_CARE` to indicate no preference.

When antialiasing is enabled, OpenGL computes the fraction of each pixel that

is covered by the point or line. The setting of the GL_LINE_SMOOTH and the GL_POINT_SMOOTH hints determine how accurate the calculation is when rendering lines and points, respectively. When the hint is set to GL_NICEST, a larger filter function may be applied causing more fragments to be generated and rendering to slow down.

If you are using RGBA rendering, OpenGL will set the alpha value according to the pixel coverage. You need to enable blending so that the incoming pixel fragment will be combined with the value already in the framebuffer, depending on the alpha value. You will probably want to set the blending factors to GL_SRC_ALPHA (source) and GL_ONE_MINUS_SRC_ALPHA (destination). You can also use GL_ONE for the destination factor to make lines a little brighter where they intersect.

Antialiasing in color index mode is trickier because you have to load the color map correctly to get primitive edges to blend with the background color. When antialiasing is enabled, the last four bits of the color index indicate the coverage value. Thus, you need to load sixteen contiguous colormap locations with a color ramp ranging from the background color to the object's color. This technique only works well when drawing wireframe images, where the lines and points typically need to be blended with a constant background. If the lines and/or points need to be blended with background polygons or images, RGBA rendering should be used.

You need to be careful when rendering antialiased lines and points with depth buffered primitives. You should draw the depth buffered primitives first and then draw the points and lines with the depth test still enabled but with depth buffer updates disabled. This way the points and lines will be 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 antialiased 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 antialiased 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 antialiased lines it is often necessary to adjust the gamma of the monitor to get the best results.

7.2 Polygon Antialiasing

Antialiasing the edges of filled polygons is similar to antialiasing points and lines. However, antialiasing polygons in color index mode isn't practical since object intersections are more prevalent and you really need to use OpenGL blending to get decent results.

To enable polygon antialiasing call glEnable with GL_POLYGON_SMOOTH.

This causes pixels on the edges of the polygon to be assigned fractional alpha values based on their coverage. Also, if you want, you can supply a value for GL_POLYGON_SMOOTH_HINT.

In order to get the polygons blended correctly when they overlap, you need to sort the polygons in front to back order. Before rendering, disable depth testing, enable blending and set the blending factors to GL_SRC_ALPHA_SATURATE (source) and GL_ONE (destination). The final color will be the sum of the destination color and the scaled source color; the scale factor is the smaller of either the incoming source alpha value or one minus the destination alpha value. This means that for a pixel with a large alpha value, successive incoming pixels have little effect on the final color because one minus the destination alpha is almost zero.

Since the accumulated coverage is stored in the color buffer, destination alpha is required for this algorithm to work. Thus you must request a visual or pixel format with destination alpha. OpenGL does not require implementations to support a destination alpha buffer so visual selection may fail.

7.3 Multisampling

Multisampling is an antialiasing method that provides high quality results. It is available as an OpenGL extension from at least one vendor. 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. Current implementations support four, eight, and sixteen samples per pixel. The method allows for full scene antialiasing 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. Antialiased lines and points using alpha coverage can be mixed with multisampling as well as the accumulation buffer antialiasing method.

7.4 Antialiasing With Textures

You can also antialias points and lines using the filtering provided by texturing. For example, to draw antialiased points, create a texture image containing a filled circle with a smooth (antialiased) boundary. Then apply the texture to the point making sure that the center of the texture is aligned with the point's coordinates and using the texture environment GL_MODULATE. 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 antialiased line segments of any width. The texture image is a filtered circle as described 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.

You can also use alpha textures to accomplish antialiasing. Simply create an image of a circle where the alpha values are one in the center and go to zero as you move from the center out to an edge. The alpha texel values would then be used to blend the point or rectangle fragments with the pixel values already in the framebuffer.

7.5 Antialiasing with Accumulation Buffer

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 taken 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 wasn't created by calling `glOrtho` or `gluOrtho2D`, then you will need to use the viewing volume extents (right, left, top, bottom) to compute scalex and scaley as follows:

```
GLfloat right, left, top, bottom;

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

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 in Table 2.

Using the accumulation buffer, you can easily trade off quality and speed. For higher quality images, simply increase the number of scenes that are accumulated. Although it is simple to antialias the scene using the accumulation buffer, it is much more computationally intensive and probably slower than the polygon antialiasing

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 2: Sample Jittering Values

method described above. Also, OpenGL does not require implementations to support an accumulation buffer, so you may not be able to select a visual or pixel format with an accumulation buffer.

8 Lighting

This section discusses varies ways of improving and refining the lighting of your scenes using OpenGL.

8.1 Phong Shading

8.1.1 Phong Highlights with Texture

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 framebuffer using a blending function which sums 100% of the source fragment and 100destination 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 `GL_ONE`, `GL_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 [37] and then use the GL_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 GL_ONE, GL_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.

8.1.2 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 `GL_EYE_LINEAR` texture generation function with an `GL_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 `GL_MODULATE`.

5. Enable the texgen functions, load the texture matrix.
6. Enable blending and set the blend function to GL_ONE, GL_ONE.
7. Disable depth buffer updates and set the depth function to GL_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 GL_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 GL_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 ambient 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 1.1 involves illuminating the scene using normal OpenGL lighting with the spotlight texture modulating this illumination.

8.1.3 Phong shading by Adaptive Tessellation

Phong highlights can also be approached with a modeling technique. The surface can be adaptively tessellated until the difference between $(\vec{H} \cdot \vec{N})^n$ terms on triangle vertices drops below a predetermined value. The advantage of this technique is that it can be done as a separate pre-processing step. The disadvantage is that it increases the complexity of the modeled object. This can be costly if:

- The model will have to be clipped by a large number of user-defined clipping planes
- The model will have tiled textures applied to it.
- The performance of the application/system is already triangle limited.

8.2 Light Maps

A light map is a texture map applied to a material to simulate the effect of a local light source. Like specular highlights, it can be used to improve the appearance of local light sources without resorting to excessive tessellation of the objects in the scene. A excellent example of an application using lightmaps is the interactive PC game Quake(tm). This game uses light maps to simulate the effects of local light sources, both stationary and moving, to great effect.

Using lightmaps usually requires a multipass algorithm, unless the objects being mapped are untextured. A texture simulating the light's effect on the object is created, then applied to one or more objects in the scene. Appropriate texture coordinates are generated, and texture transformations can be used to position the light, and create moving or changing light effects. Multiple light sources can be generated with a combination of more complex texture maps and/or more passes to the algorithm.

Light maps are often luminance textures, which are applied to the object using `GL_MODULATE` as the value for `GL_TEXTURE_ENV_MODE`. Colored lights can also be simulated by using an RGB texture.

Light maps can often produce satisfactory lighting effects at lower resolutions than normal textures. It is often not necessary to produce MIPmaps; choosing `GL_LINEAR` for the minification and magnification filters is often sufficient. Of course, the minimum quality of the lighting effect is a function of the intended application.

8.2.1 2D Texture Light Maps

A 2D light map is a texture map applied to the surfaces of a scene, modulating the intensity of the surfaces to simulate the effects of a local light. If the surface is already textured, then applying the light map becomes a multipass operation, modulating the intensity of a surface detail texture.

A 2D light map can be generated analytically, creating a bright spot in luminance or color values that drops off appropriately with increasing distance from the light center. As with other lighting equations, a quadratic drop off, modified with linear and constant terms can be used to simulate a variety of lights, depending on the area of the emitting source.

Since generating new textures takes time and consumes valuable texture memory, it is a good strategy to create a few canonical light maps, based on intensity drop-off characteristics and color, then use them for a number of different lights by transforming the texture coordinates. If the light source is isotropic, then simple translations and scales can be used to position the light appropriately on the surface, while scales can be used to adjust the size of the lighting effect, simulating different sizes of lights and distance from the lighted surface.

In order to apply a light map to a surface properly, the position of the light in the scene must be projected onto each surface of interest. This position shows where the bright spot will be. The perpendicular distance of the light from the surface can be used to adjust the bright spot size and brightness. One approach is to generate texture coordinates, orienting the generating planes with each surface of interest, then translating and scaling the texture matrix to position the light on the surface. This process is repeated for every surface affected by the light.

In order to repeat this process for multiple lights (without resorting to a multilight lightmap) or to light textured surfaces, the lighting must be done as a series of passes. This can be done two ways. The more straightforward way is to blend the entire scene. The other way is to blend together the surface texture and light maps to create a texture for each surface. This texture will represent the contributions of the surface texture and all lightmaps affecting its surface. The merged texture is then applied to the surface. Although more involved, the second method produces a higher quality result.

For each surface:

1. Transform the surface so that it is perpendicular to the direction of view (maximize its visible surface). Scale the image so that its area in pixels matches the desired size of the final texture.

2. Render the transformed surface into the frame buffer (this can be done in the back buffer). If it is textured, render it with the surface texture.

3. Re-render the surface, using the appropriate light map. Adjust the GL_EYE_PLANE equations and the texture transform to position the light correctly on the surface. Use the appropriate blend function.

4. Repeat the previous step with each light visible to the surface.

5. Copy the image into a texture using glCopyTexImage2D.

6. When you've created textures for all lit surfaces, render the scene using the new textures.

Since switching between textures must be done quickly, and lightmap textures tend to be small, use texture objects to switch between different light maps and surface textures to improve performance.

With either approach, the blending is a modulation of the colors of the existing texture. This can be done by rendering with the blend function (GL_ZERO, GL_SRC_COLOR). If the light map is composed of luminance values than the individual destination color components will be scaled equally, if the light map represents a colored light, then the color components of the destination will be scaled by the red, green, and blue components of the light map texel values.

Note that each modulation pass attenuates the surface color. The results will become increasingly dim. If surfaces require a large number of lights, the dynamic range of light maps can be compressed to avoid excessive darkening. Instead of ranging from 1.0 (full light) to 0.0 (no light), They can range from 1.0 (full light) to 0.5 or 0.75 (no light). The no light value can be adjusted as a function of the number of lights in the scene.

Here are the steps for using 2D Light Maps:

1. Create the 2D light data. "Canonical lights" can be defined at the center of the texture, with the intensity dropping off in a realistic fashion towards the edges. In order to avoid artifacts, make sure the intensity of the light field is the same at all the edges of the texture volume.

2. Define a 2D texture, using GL_REPEAT for the wrap values in S, T, and R. Minification and magnification should be GL_LINEAR to make the changes in intensity smoother. For performance reasons, make this texture a texture object.

3. Render the scene without the lightmap, using surface textures as appropriate.

4. For each light in the scene:

   (a) For each surface in the scene:

i. Cull surfaces that can't "see" the current light.
ii. Find the plane of the surface.
iii. Align the GL_EYE_PLANE for GL_S and GL_T with the surface plane.
iv. Scale and translate the texture coordinates to position and size the light on the surface.
v. Render the surface using the appropriate blend function and lightmap texture.

An alternative to simple light maps is to use projective textures to draw light sources. This is a good approach when doing spotlight effects. It's not as useful for isotropic light sources, since you'll have to tile your projections to make the light shine in all directions. See the projective texture description 8.1.1 and 5.8 for more details.

8.2.2 3D Texture Light Maps

3D Textures can also be used as light maps. One or more light sources are represented in 3D data, then the 3D texture is applied to the entire scene. The main advantage of using 3D textures for light maps is that it's easy to calculate the proper texture coordinates. The textured light source can be positioned globally with the appropriate texture transformations then the scene is rendered, using glTexGen to generate the proper S, T, and R coordinates.

The light source can be moved by changing the texture matrix. The resolution of the light field is dependent on the texture resolution.

A useful approach is to define a canonical light field in 3D texture data, then use it to represent multiple lights at different positions and sizes by applying texture translations and scales to shift and resize the light. Multiple lights can be simulated by accumulating the results of each light source on the scene.

To ensure that the light source can be shifted easily, set GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, and GL_TEXTURE_WRAP_R_EXT to GL_REPEAT. Then the light can be shifted to any location in the scene. Be sure that the texel values in the light map are the same at all boundaries of the texture; otherwise you'll be able to see the edges of the texture as vertical and horizontal "shadows" in the scene.

Although it is uncommon, some types of light fields would be very hard to do without 3D textures. A complex light source, whose brightness and range varies as a function of distance from the light source could be best done with a 3D texture. An example might be a "disco ball" effect where a light source has beams emanating out from the center, with some beams shining farther than others. A complex

light source could be made more impressive by combining light maps with volume visualization techniques. For example the light beams could be made visible in fog.

The light source itself can be a simple piece of geometry textured with the rest of the scene. Since it is at the source of the textured light, it will be textured brightly.

For better realism, good lighting effects should be combined with the shadowing techniques described in Section 9.4.

Procedure:

1. Create the 3D light data. A "canonical light" can be defined at the center of the texture volume, with the intensity dropping off in a realistic fashion towards the edges. In order to avoid artifacts, make sure the intensity of the light field is the same at all the edges of the texture volume.

2. Define a 3D texture, using `GL_REPEAT` for the wrap values in S, T, and R. Minification and magnification should be `GL_LINEAR` to make the changes in intensity smoother.

3. Render the scene without the lightmap, using surface textures as appropriate.

4. Define planes in eye space so that `glTexGen` will cause the texture to span the visible scene.

5. If you have textured surfaces, adding a lightmap becomes a multipass technique. Use the appropriate blending function to modulate the surface color.

6. Render the image with the light map, and texgen enabled. Use the appropriate texture transform to position and scale the light source correctly.

7. Repeat steps 1-2 and 4-6 for each light source.

There are disadvantages to using 3D light maps:

- 3D textures are not widely supported yet, so your application will not be as portable.

- 3D textures use a lot of texture memory. 2D textures are more efficient for light maps.

8.3 Bump Mapping with Textures

Bump mapping [6], like texture mapping, is a technique to add more realism to synthetic images without adding a lot of geometry. Texture mapping adds realism by

attaching images to geometric surfaces. Bump mapping adds per-pixel surface relief shading, increasing the apparent complexity of the surface.

Surfaces that should have a patterned roughness are good candidates for bump mapping. Examples include oranges, strawberries, stucco, wood, etc.

A bump map is an array of values that represent an object's height variations on a small scale. A custom renderer is used to map these height values into changes in the local surface normal. These perturbed normals are combined with the surface normal, and the results are used to evaluate the lighting equation at each pixel.

The technique described here uses texture maps to generate bump mapping effects without requiring a custom renderer [1] [36]. This multipass algorithm is an extension and refinement of texture embossing [42].

The first derivative of the height values of the bump map can found by the following process:

1. Render the image as a texture.
2. Shift the texture coordinates at the vertices.
3. Re-render the image as a texture, subtracting from the first image.

Consider a one dimensional bump map for simplicity. The map only varies as a function of S. Assuming that the height values of the bump map can be represented as a height function f(s), then the three step process above would be like doing the following: $f(s) - f(s + shift)$. If the shift was by one texel in S, you'd have $f(s) - f(s + \frac{1}{w})$, where w is the width of the texture in texels. This is a different form of $\frac{f(s) - f(s+1)}{1}$ which is just the basic derivative formula. So shifting and subtracting results in the first derivative of $f(s)$, $f'(s)$.

In the two dimensional case, the height function is $f(s,t)$, and shifting and subtracting creates a directional derivative of $f(s,t)$. This technique is used to create embossed images.

With more precise shifting of the texture coordinates, we can get general bump mapping from this technique.

8.3.1 Tangent Space

In order to accurately shift, the light source direction $\vec{L}$ must be rotated into *tangent space*. Tangent space has 3 perpendicular axis, T, B and N. T, the tangent vector, is parallel to the direction of increasing S or T on a parametric surface. N, the normal vector, is perpendicular to the local surface. B, the binormal, is perpendicular to both N and T, and like T, also lies on the surface. They can be thought of as forming a coordinate system that is attached to surface, keeping the T and B vectors pointing

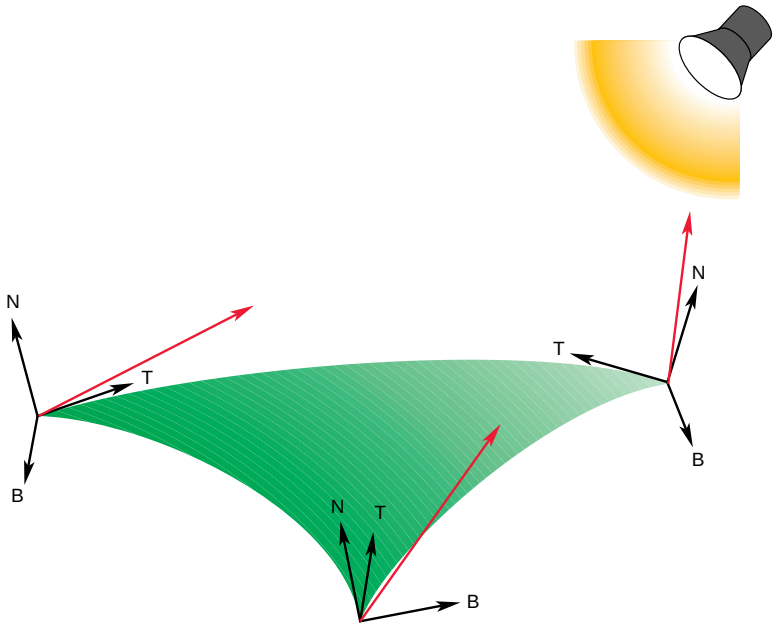


Figure 20. Tangent Space Defined at Polygon Vertices

along the tangent of the surface, and N pointing away. If the surface is curved, the tangent space orientation changes at every point on the surface.

In order to create a tangent space for a surface, it must be mapped parametrically. But since this technique requires applying a 2D texture map to the surface, the object must already be parametrically mapped in S and T. If the surface is already mapped with a surface detail texture, the S and T coordinates of that mapping can be reused. If it is a NURBS surface, the S and T values of that mapping can be used. The only requirement for bump mapping to work is that the parametric mapping be consistent on the polygon. Of course, to avoid "cracking" between polygons, the mapping should be consistent across the entire surface.

The light source must be rotated into tangent space at each vertex of the polygon. To find the tangent space vectors at a vertex, use the vertex normal for N, find the tangent axis by finding the vector direction of increasing S in the object's coordinate system (the direction of the texture's S axis in the object's space). You could use the texture's T axis as the tangent axis instead if it is more convenient. Find B by computing the cross product of N and T. The normalized values of these vectors

can be used to create a rotation matrix:

$$\begin{bmatrix} Tx & Ty & Tz & 0 \\ Bx & By & Bz & 0 \\ Nx & Ny & Nz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This matrix rotates the T vector, defined in object space, into the X axis of tangent space, the B vector into the Y axis, and the normal vector into the Z axis. It rotates a vector from object space into tangent space. If the T, B and N vectors are defined in eye space, then it converts from eye space to tangent space. For all non-planar surfaces, this matrix will differ at each vertex of the polygon.

Now you can apply this matrix to the light direction vector $\vec{L}$, transforming it into tangent space at each vertex. Use the transformed X and Y components of the light vector to shift the texture coordinates at the vertex.

The resulting image, after shifting and subtracting is part of $\vec{N} \cdot \vec{L}$, computed in tangent space at every texel. In order to get the complete dot product, you need to add in the rotated Z component of the light vector. This is done as a separate pass, blending the results with the previous image, but adding, not subtracting this time. It turns out that this third component is the same as adding in the Gouraud shaded version of the polygon to the textured one.

So the steps for diffuse bump mapping are:

1. Render the polygon with the bump map textured on it. Since the bump map modifies the polygon color, you can get the diffuse color you want by coloring the polygon with k_d.
2. Find $\hat{N}$, $\hat{T}$ and $\hat{B}$ at each vertex.
3. Use the vectors to create a rotation matrix.
4. Use the matrix to rotate the light vector $\hat{L}$ into tangent space.
5. Use the rotated X and Y components of $\hat{L}$ to shift the S and T texture coordinates at each polygon vertex.
6. Re-render the bump map textured polygon using the shifted texture coordinates.
7. Subtract the second image from the first.
8. Render the polygon Gouraud shaded with no bump map texture.
9. Add this image to result.

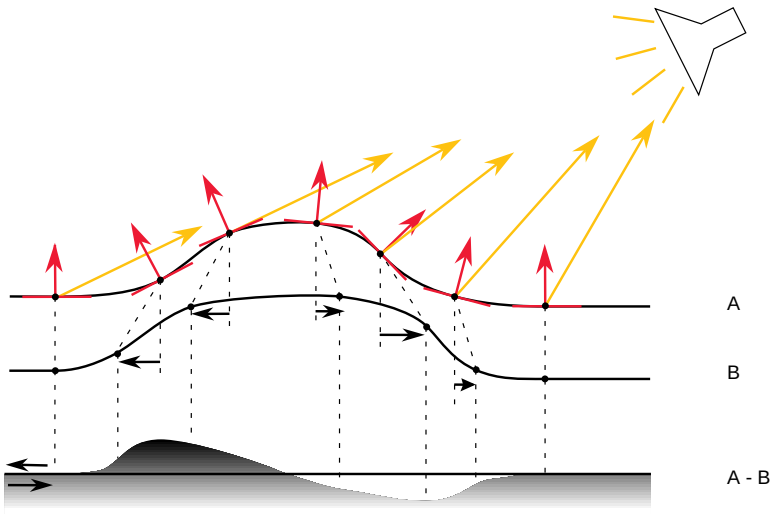


Figure 21. Shifting Bump Mapping to Create Normal Components

In order to improve accuracy, this process can be done using the accumulation buffer. The bump mapped objects in the scene are rendered with the bump map, re-rendered with the shifted bump map and accumulated with a negative weight, then re-rendered again using Gouraud shading and no bump map texture, accumulated normally.

The process can be extended to find bump mapped specular highlights. The process is repeated, this time using the halfway vector ($\vec{H}$) instead of the light vector. The halfway vector is computed by averaging the light and viewer vectors $\frac{\hat{L}+\hat{V}}{2}$. Here are the steps for finding specular bump mapping:

1. Render the polygon with the bump map textured on it.
2. Find $\hat{N}$, $\hat{T}$ and $\hat{B}$ at each vertex.
3. Use the vectors to create a rotation matrix.
4. Use the matrix to rotate the halfway vector $\hat{H}$ into tangent space.
5. Use the rotated X and Y components of $\hat{H}$ to shift the S and T texture coordinates at each polygon vertex.

6. Re-render the bump map textured polygon using the shifted texture coordinates.

7. Subtract the second image from the first.

8. Render the polygon Gouraud shaded with no bump map texture, this time use $\hat{H}$ instead of $\hat{L}$. Use a polygon whose color is equal to the specular color you want, k_s.

9. Now you have $(\hat{H} \cdot \hat{N})$, but you want $(\hat{H} \cdot \hat{N})^n$ To raise the result to a power, you can load power function values into the texture color table, using `glColorTableSGI` with `GL_TEXTURE_COLOR_TABLE_SGI` as its target, then enabling `GL_TEXTURE_COLOR_TABLE_SGI`. With the color lookup table loaded and enabled, when you texture and blend the specular contribution to the result, the texture filtering will raise the specular dot product to the proper power. If you don't have this extension, then you can process the texel values on the host, or limit yourself to non-bump mapped specular hightlights.

10. Add this image to result.

Combine the two images together to get both contributions in the image.

8.3.2 Going for higher quality

The previous technique renders the entire scene multiple times. If very high quality is important, the texture itself can be processed separately, then applied to the scene as a final step. The previous technique yields lower quality results where the texture is less perpendicular to the line of sight in the image, due to the object geometry. If the texture is processed before being applied to the image, we avoid this problem.

To process the texture separately, the vertices of the object must be mapped to a square grid. The rest of the steps are the same, because the relationship between light source and the vertex normals hasn't changed. When the new texture map has been created, copy it back into texture memory, and use it to render the object.

8.4 Blending

If you choose not to use the accumulation buffer, acceptable results can be obtained by blending. Because of the subtraction step, you'll have to remap the color values to avoid negative results. Since the image values range from 0 to 1, the range of values after subtraction can be -1 (0 - 1) to 1 (1 - 0).

Scale and bias the bump map values to remap the results to the 0 to 1 range. Once you've made all three passes, it is safe to remap the values back to their original 0 to 1 range. This scaling and biasing, combined with less bits of color precision, makes this method inferior to using the accumulation buffer.

8.4.1 Why does this work?

By shifting and subtracting the bump map, you're finding the directional derivative of the bump map's height function.

By rotating the light vector into tangent space, then using the X and Y components for the shift values, you're finding the component of the perturbed normal vector aligned with the light. In tangent space, the unperturbed normal is a unit vector along the Z axis. When the shifted values are non-zero, they represent the magnitude of the component of the perturbed normal in the direction of the light source. Since the perturbed normal component is parallel to the light source vector (in tangent space), the dot product of this component with the light reduces to a scale operation, which is what a texture map with the texture environment set to modulate does.

Since the perturbed normal is relative to the smooth surface normal, we take the smoothed normal contribution into account when we add in the Gouraud shaded polygon.

There is an assumption that the perturbed normal is not much different from the smoothed surface unit normal, so that the length of the perturbed normal is not much different from one. If this assumption wasn't true, we'd have to create and modulate in an extra texture that would renormalize the perturbed normal. This can be done, at the cost of an extra texturing pass, if more accuracy is needed.

8.4.2 Limitations

Although this technique does correctly bump map the surface efficiently, there are limitations to its accuracy.

Bump map Sampling The bump map height function is not continuous, but is sampled into the texture. The resolution of the texture affects how faithfully the bump map is represented. Increasing the size of the bump map texture can improve the sampling of the high frequency height components.

Texture Resolution The shifting and subtraction steps produce the directional derivative. Since this is a forward differencing technique, the highest frequency component of the bump map increases as the shift is made smaller. As the shift is made smaller, more demands are made of the texture coordinate

precision. The shift can become smaller than the texture filtering implementation can handle, leading to noise and aliases effects. A good starting point is to size the shift components so their vector magnitude is a single texel.

Surface Curvature The tangent coordinate axes are different at each point on a curved surface. This technique approximates this by finding the tangent space transforms at each vertex. Texture mapping interpolates the different shift values from each vertex across the polygon. For polygons with very different vertex normals, this approximation can break down. A solution would be to subdivide the polygons until their vertex normals are parallel to within some error limit.

Maximum Bump map Slope The bump map normals used in this technique are good approximations if the bump map slope is small. If there are steep tangents in the bump map, the assumption that the perturbed normal is length one becomes inaccurate, and the hightlights appear too bright. This can be corrected by creating a fourth pass, using a modulating texture derived from the original bump map. Each value of the texel is one over the length of the perturbed normal: $1/\sqrt{\frac{\partial f}{\partial u}^2 + \frac{\partial f}{\partial v}^2 + 1}$

8.5 Choosing Material Properties

OpenGL provides a full lighting model to help produce realistic objects. The library provides no guidance, however, on finding the proper lighting material parameters to simulate specific materials. This section categorizes common materials, provides some guidance for choosing representative material properties, and provides a table of material properties for common materials.

8.5.1 Modeling Material Type

Material properties are modelled with the following OpenGL parameters:

`GL_AMBIENT` How ambient light reflects from the material surface. This is an RGBA color vector. The magnitude of each component indicates how much the light of that component is being reflected.

`GL_DIFFUSE` How diffuse reflection from light sources reflect from the material surface. This is an RGBA color vector. The magnitude of each component indicates how much the light of that component is being reflected.

`GL_SPECULAR` How specular reflection from a light source reflects from the material. This is an RGBA color vector. The magnitude of each component indicates how much the light of that component is being reflected.

`GL_EMISSION` How much of what color is being emitted from this object. This is an RGBA color vector. The magnitude of each component indicates how much light of that component is glowing from the material. Since this parameter is only useful for glowing objects, we'll ignore it in this section.

`GL_SHININESS` How mirror-like the specular reflection is from this material. This is a single integer. The larger the number, the more rapidly the specular reflection drops off as the viewing angle diverges from the reflection vector.

For lighting purposes, materials can be described by the type of material, and the smoothness of its surface. Material type is simulated by the relationship between color components of the `GL_AMBIENT`, `GL_DIFFUSE` and `GL_SPECULAR` parameters. Surface smoothness is simulated by the overall magnitude of the `GL_AMBIENT`, `GL_DIFFUSE` and `GL_SPECULAR` parameters, and the value of `GL_SHININESS`. As the magnitude of these components get closer to one, and the `GL_SHININESS` value increases, the material appears to have a smoother surface.

For lighting purposes, material type can be divided into four categories: dielectrics, metals, composites, and other materials.

Dielectrics These are the most common category. These are non-conductive materials, which don't have free electrons. The result is that dieletrics have low reflectivity, and have a reflectivity that is independent of light color. Because they don't interact with the light much, dieletrics tend to be transparent. The ambient, diffuse and specular colors tend to be the same.

Powdered dieletrics tend to look white because of the high surface area between the dielectric and the surrounding air. Because of this high surface area, they also tend to reflect diffusely.

Metals Metals are conductive and have free electrons. As a result, metals are opaque and tend to be very reflective, and their ambient, diffuse, and specular colors tend to be the same. How the free electrons are excited by light at different wavelengths determines the color of the metal. Materials like steel and nickel have nearly the same response over all visible wavelengths, resulting in a grayish reflection. Copper and gold, on the other hand, reflect long wavelengths more strongly than short ones, giving them their reddish and yellowish colors.

The color of light reflected from metals is also a function of incident and exiting light directions. This can't be modeled accurately with the OpenGL lighting model,

compromising the metallic look of objects. However, a modified form of environment mapping (such as the OpenGL sphere mapping) can be used to approximate the proper visual effect.

Composite Materials Common composites, like plastic and paint, are composed of a dielectric binder with metal pigments suspended in them. As a result, they combine the reflective properties of metals and dieletrics. Their specular reflection is dielectric, their diffuse reflection is like metal.

Other Materials Other materials that don't fit into the above categories are materials such as thin films, and other exotics.

8.5.2 Modeling Material Smoothness

As mentioned before, the apparent smoothness of a material is a function of how strongly it reflects and the size of the specular highlight. This is affected by the overall magnitude of the `GL_AMBIENT`, `GL_DIFFUSE` and `GL_SPECULAR` parameters, and the value of `GL_SHININESS`. Here are some heuristics that describe useful relationships between the magnitudes of these parameters:

1. The spectral color of the `GL_AMBIENT` and `GL_DIFFUSE` parameters should be the same.
2. The magnitudes of `GL_DIFFUSE` and `GL_SPECULAR` should sum to a value close to one. This helps prevent color value overflow.
3. The value of `GL_SHININESS` should increase as the magnitude of `GL_SPECULAR` approaches one.

No promise is made that these relationships, or the values in Table 3 will provide a perfect imitation of a given material. The empirical model used by OpenGL emphasizes performance, not physical exactness.

For an excellent description of material properties, see [23] for more information.

Material	GL_AMBIENT	GL_DIFFUSE	GL_SPECULAR	GL_SHININESS
Brass	0.329412 0.223529 0.027451 1.0	0.780392 0.568627 0.113725 1.0	0.992157 0.941176 0.807843 1.0	27.8974
Bronze	0.2125 0.1275 0.054 1.0	0.714 0.4284 0.18144 1.0	0.393548 0.271906 0.166721 1.0	25.6
Polished Bronze	0.25 0.148 0.06475 1.0	0.4 0.2368 0.1036 1.0	0.774597 0.458561 0.200621 1.0	76.8
Chrome	0.25 0.25 0.25 1.0	0.4 0.4 0.4 1.0	0.774597 0.774597 0.774597 1.0	76.8
Copper	0.19125 0.0735 0.0225 1.0	0.7038 0.27048 0.0828 1.0	0.256777 0.137622 0.086014 1.0	12.8
Polished Copper	0.2295 0.08825 0.0275 1.0	0.5508 0.2118 0.066 1.0	0.580594 0.223257 0.0695701 1.0	51.2
Gold	0.24725 0.1995 0.0745 1.0	0.75164 0.60648 0.22648 1.0	0.628281 0.555802 0.366065 1.0	51.2
Polished Gold	0.24725 0.2245 0.0645 1.0	0.34615 0.3143 0.0903 1.0	0.797357 0.723991 0.208006 1.0	83.2
Pewter	0.105882 0.058824 0.113725 1.0	0.427451 0.470588 0.541176 1.0	0.333333 0.333333 0.521569 1.0	9.84615

Table 3: Parameters for common materials

Material	GL_AMBIENT	GL_DIFFUSE	GL_SPECULAR	GL_SHININESS
Silver	0.19225 0.19225 0.19225 1.0	0.50754 0.50754 0.50754 1.0	0.508273 0.508273 0.508273 1.0	51.2
Polished Silver	0.23125 0.23125 0.23125 1.0	0.2775 0.2775 0.2775 1.0	0.773911 0.773911 0.773911 1.0	89.6
Emerald	0.0215 0.1745 0.0215 0.55	0.07568 0.61424 0.07568 0.55	0.633 0.727811 0.633 0.55	76.8
Jade	0.135 0.2225 0.1575 0.95	0.54 0.89 0.63 0.95	0.316228 0.316228 0.316228 0.95	12.8
Obsidian	0.05375 0.05 0.06625 0.82	0.18275 0.17 0.22525 0.82	0.332741 0.328634 0.346435 0.82	38.4
Pearl	0.25 0.20725 0.20725 0.922	1.0 0.829 0.829 0.922	0.296648 0.296648 0.296648 0.922	11.264
Ruby	0.1745 0.01175 0.01175 0.55	0.61424 0.04136 0.04136 0.55	0.727811 0.626959 0.626959 0.55	76.8
Turquoise	0.1 0.18725 0.1745 0.8	0.396 0.74151 0.69102 0.8	0.297254 0.30829 0.306678 0.8	12.8
Black Plastic	0.0 0.0 0.0 1.0	0.01 0.01 0.01 1.0	0.50 0.50 0.50 1.0	32
Black Rubber	0.02 0.02 0.02 1.0	0.01 0.01 0.01 1.0	0.4 0.4 0.4 1.0	10

9 Scene Realism

9.1 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, 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.

9.2 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

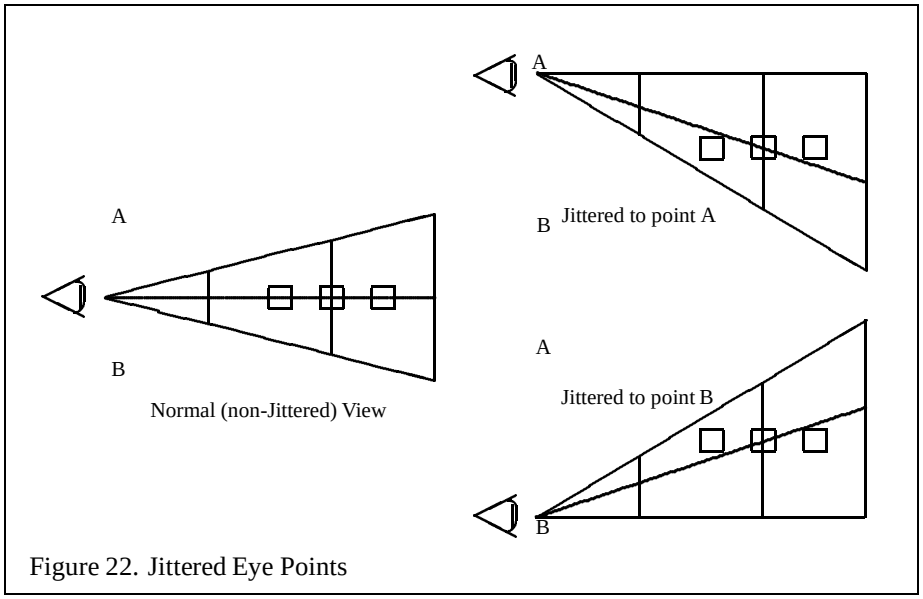


Figure 22. Jittered Eye Points

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 changes 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)
{
    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 sets the modelview matrix to the identity, and loads it with a translation. 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.

9.3 Reflections and Refractions

In both rendering and interactive computer graphics, substantial effort has been devoted to the modeling of reflected and refracted light. This is not surprising – almost all the light perceived in the world is reflected. In this section, we will describe several ways to create the effects of reflection and refraction using OpenGL. We will begin with a very brief review of the relevant physics and give pointers to more detailed descriptions.

From elementary physics, we know that the angle of reflection of a ray is equal to the angle of incidence of the ray (Figure 23). This property is known as the *Law of Reflection*.[10]. The reflected ray lies in the plane defined by the incident ray and the surface normal.

Refraction is defined as the "change in the direction of travel as light passes from one medium to another."[10]. This change in direction is caused by the difference in the speed of light traveling through the two mediums. The refractivity of a material is characterized by the *index of refraction* of the material, or the ratio of the speed of light in the material to the speed of light in a vacuum.[10].

The direction of a light ray after it passes from one medium to another is computed from the direction of the incident ray, the normal of the surface at the intersection of the incident ray, and the indices of refraction of the two materials. The

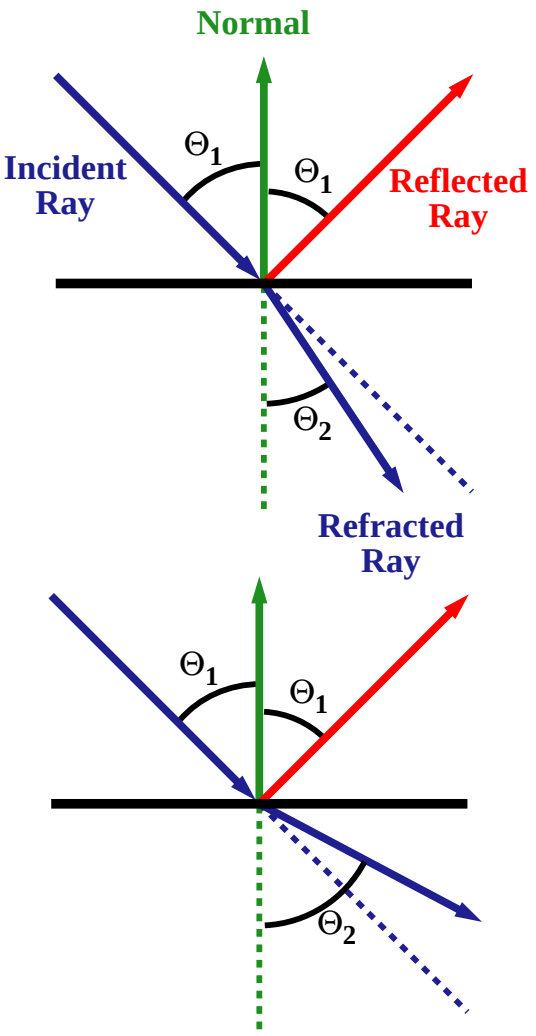


Figure 23. Reflection and refraction. The image on the top shows transmission from a medium with a lower to a higher index of refraction; the image on the bottom shows transmission from higher to lower.

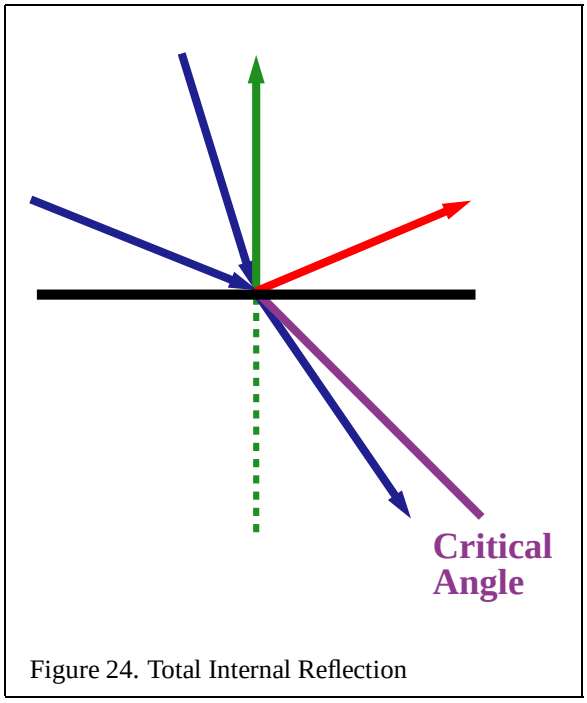


Figure 24. Total Internal Reflection

behavior is shown in Figure 23. The first medium through which the ray passes has an index of refraction n_1 and the second has an index of refraction n_2. The angle of incidence Θ_1 is the angle between the incident ray and the surface normal. The refracted ray forms the angle Θ_2 with the normal. The incident and refracted rays are coplanar. The relationship between the angle of incidence and the angle of refraction is stated as *Snell's Law*[10]:

$$n_1 \cos \Theta_1 = n_2 \cos \Theta_2 \tag{1}$$

If $n_1 > n_2$ (light is passing from a more refractive material to a less refractive material), past some critical angle the incident ray will be bent so far that it will not cross the boundary. This phenomenon is known as *total internal reflection* and is illustrated in Figure 24.[10]

When a ray hits a surface, some light is reflected off the surface and some is transmitted. The weighting of the transmitted and reflected light is determined by the *Fresnel equations*.

More details about reflection and refraction can be gleaned from most college physics books. For more details on the reflection and transmission of light from a computer graphics perspective, the reader may consult one of several general computer graphics books or books on radiosity or ray tracing. The following books may prove helpful:

- Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Harcourt Brace & Company, 1993.
- Andrew S. Glassner. *Principles of Digital Image Synthesis*. Mogran Kaufman Publishers, Inc., 1995.
- Roy Hall. *Illumination and Color in Computer Generated Imagery*. Springer-Verlag, 1989.

9.3.1 Planar Reflectors

In this section, we will discuss the modeling of planar reflective surfaces. Two techniques are discussed: a technique which uses the stencil buffer to draw the reflected geometry in the proper location and a technique which uses texture mapping to make an image of the reflected geometry which is then texture mapped onto the reflective polygon. Both techniques construct the scene in two (or more) passes.

Planar Reflections and Refractions using the Stencil Buffer The effects of specular reflection can be approximated by a two-pass technique using the stencil

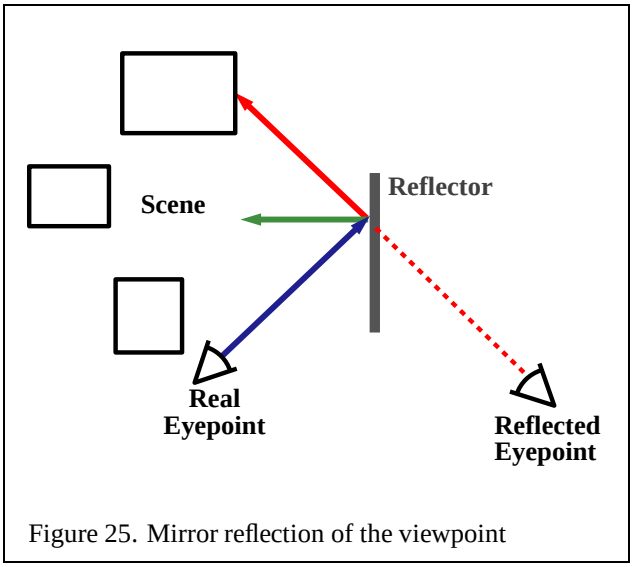


Figure 25. Mirror reflection of the viewpoint

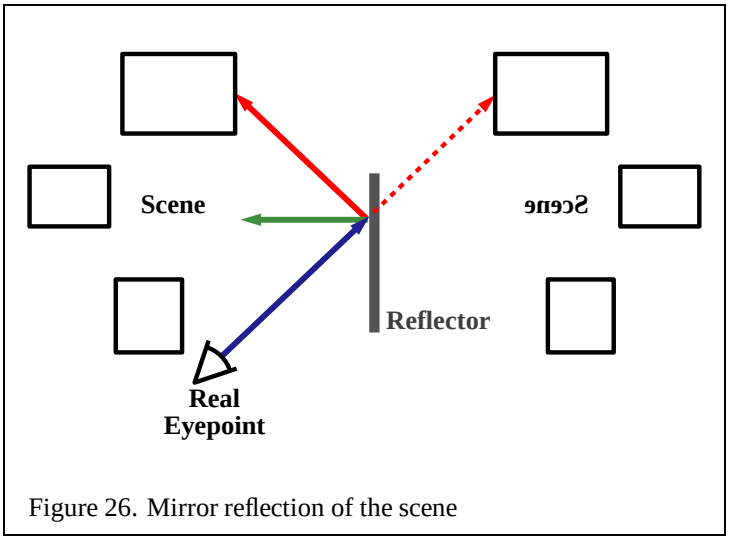


Figure 26. Mirror reflection of the scene

buffer. During the first pass, we render the reflected image of the scene. During the second pass, we render the non-reflected view of the scene, using the stencil buffer to prevent the reflected image from being drawn over.

As an example, consider a model of a room with a mirror on one wall. We compute the plane containing the mirror and define an eyepoint from which we wish to render the scene. During the first pass, we place the eyepoint at the desired location (using a `gluLookAt` command or something similar). Next, we draw the scene as it looks reflected through the plane containing the mirror. This can be envisioned in two ways, shown in Figures 25 and 26. In the first illustration, we reflect the viewpoint. In the second illustration, we reflect the scene. The ways of considering the problem are equivalent. We present both here since reflecting the viewpoint will tie into the next section, but many people seem to find reflecting the scene more intuitive. The sequence of steps for the first pass is as follows:

1. Initialize the modelview and projection matrices to the identity (`glLoadIdentity`).
2. Set up a projection matrix using the `glFrustum` command.
3. Set up the "real" eyepoint at the desired position using a `gluLookAt(c)`ommand (or something similar).
4. Reflect the viewing frustum (or the scene) through the plane containing the reflector by computing a reflection matrix and combining it with the current modelview or projection matrices using the `glMultMatrix` command.
5. Draw the scene.
6. Move the eyepoint back to its "real" position.

Objects drawn in the first pass look as they would when seen in the mirror, except that we ignore the fact that the mirror may not fill the entire field of view. That is to say, we imagine that the entire plane containing the mirror is reflective, but in reality the mirror does not cover the entire plane. Parts of the scene may be drawn which will not be visible. For example, the lowest box in the scene in Figure 26 is drawn, but its reflection is not visible in the mirror. We'll fix this in the second pass.

When we render from the reflected eyepoint, points on the plane through which we reflect maintain the same position in eyespace as when we render from the original eyepoint. For example, corners of the reflective polygon are in the same location when viewed from the reflected eyepoint as from the original viewpoint. This may seem more believable if one imagines that we are reflecting the scene, instead of the eyepoint.

One implementation problem during the first pass is that we should not draw the mirror or it will obscure our reflected image. This problem may be solved by

backface culling, or by having the graphics application recognize the mirror (and objects in the same plane as the mirror).

We may wish to produce a magnified or minified reflection by moving the reflected viewpoint backwards or forwards along its line of sight. If the position is the same distance as the eye point from the mirror then an image of the same scale will result.

We start the second pass by setting the eyepoint up at the "real" location. Next, we draw the mirror polygon. We wish to mask out portions of the reflected scene which we drew in the first pass, but which should not be visible. This is accomplished using the stencil buffer. First, we clear the stencil and depth buffers. Next, we draw the mirror polygon into the stencil buffer and depth buffers, setting the stencil value to 1. We may or may not wish to render the mirror polygon to the color buffers at this point. If we do, the mirror must not be opaque or it will completely obscure our reflected scene. We can give the appearance of a dirty, not purely reflective, mirror by drawing it using one of the transparency techniques discussed in Section 10. After drawing the mirror, we configure the stencil test to pass where ever the stencil buffer value is not equal to 1. We then clear the color buffers, which erases all parts of the reflected scene except those in the mirror polygon. After the clear, we disable the stencil test and draw the scene. The list of steps for the second pass is:

1. Clear the stencil and depth buffers (`glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`).
2. Configure the stencil buffer such that a `1` will be stored at each pixel touched by a polygon:

```
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
glStencilFunc(GL_ALWAYS, 1, 1);
glEnable(GL_STENCIL_TEST);
```

3. Disable drawing into the color buffers (`glColorMask(0, 0, 0, 0)`).
4. Draw the mirror polygon.
5. Reconfigure the stencil test:

```
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
glStencilFunc(GL_NOTEQUAL);
```

6. Draw the scene.
7. Disable the stencil test (`glDisable(GL_STENCIL_TEST)`).

The frame is now complete.

Planar Reflections using Texture Mapping A technique similar to the stencil buffer technique uses texture mapping. The first pass is identical to the first pass of the previous technique: we draw the reflected scene. After drawing the scene, we copy the image into a texture (using the `glCopyTexImage2D` command). During the second pass, this texture is mapped onto the reflective polygon. The sequence of steps for the second pass is as follows:

1. Position the viewer at the "real" eyepoint.
2. Draw the non-reflective objects in the scene.
3. Bind the texture containing the reflected image.
4. Draw the reflective object with the appropriate texture coordinates.

The texture coordinates at the vertices of the reflective object must be in the same location as the vertices of the reflective object in the texture. These coordinates may be computed by figuring the projection of the corners of the object into the viewing plane used to compute the reflection map (the command `gluProject` may prove helpful). Alternately, the texture matrix can be loaded with the composite modelview and projection matrices and postmultiplied by a scale of 1 divided by the size in pixels of the region used to compute the texture. The texture coordinates would then be the model coordinates of the vertices.

The texture mapping technique may be more efficient on some systems. Also, we may be able to use a reflection texture during several frames (see below).

Interreflections Either the stencil technique or the texture mapping technique may be used to model scenes with interreflections. Each algorithm uses additional passes for each "bounce" that the light takes, stopping when the reflected image added by the pass is too small to be significant.

Using the stencil technique, we draw the reflected image with the most "bounces" from the viewpoint first. We compute the viewpoint for this pass by repeatedly reflecting the viewpoint through the reflective polygons. On each pass, we draw the scene, move the viewpoint to the next position, and draw the scene using the stencil buffer to mask the reflective polygons from the previous passes.

Using the texture technique, we first create textures for each of the reflective objects. We then initialize the textures to some known value (choice of this value will be discussed below). Next, we iterate over the primitives, drawing the scene for each one and copying the results to the primitive's reflection map as described above. We repeat this process until we determine that the additional passes are not having a significant effect.

The choice of the initial reflection map values can have an effect on the number of passes required. The initial reflection value will generally appear as a smaller part

of the picture on each of the passes. We stop the iteration when the initial reflection is small enough that the viewer will not notice that it is not correct. By setting the initial reflection to something reasonable, we can achieve this state earlier. A good initial guess is to set the map to the average color of the scene. In a multiframe application with moving objects or a moving viewpoint, we could leave the reflection map with the contents from the previous frame. This use of previous results is one of the advantages of the texture mapping technique.

9.3.2 Sphere Mapping

Sphere mapping is an implementation of *environment mapping*. Environment mapping is a computer graphics technique which uses a two-dimensional image (or images) containing the incident illumination from every direction at a given point. When rendering, the light from the point is computed as a function of the outgoing direction and the environment map. The outgoing direction is used to choose one or more incoming directions, or points in the environment map, which are used to compute the outgoing color.[35] In general, only one environment map point is used for each outgoing ray, resulting in a perfect specular reflection.

In rendering, we often use a single environment map for an entire object by assuming that the single environment map is a reasonable approximation of the environment map which would be computed at each point on the object. This approximation is correct if the object is a sphere and the viewer and other objects in the scene are infinitely far away. The approximation becomes less correct if the object has interreflections (i.e., it's not convex) and if the viewer and other objects are not at infinity. In interactive polygonal rendering, we make the additional assumption that the indices into the environment map may be computed at each vertex and linearly interpolated over each polygon. In spite of these simplifying assumptions, results in practice are generally quite good.

While rendering, we compute the outgoing direction as a function of the eyepoint and the normal at the surface. We can use environment maps to represent any effect that depends only upon the viewing direction and the surface normal. These effects include specular and directional diffuse reflection, refraction, and Phong lighting. We will discuss several of these effects in the context of OpenGL's sphere mapping capability.

Sphere mapping is a type of environment mapping in which the irradiance image is equivalent to that which would be seen in a perfectly reflective hemisphere when viewed using an orthographic projection.[35] This concept is illustrated in Figure 27. The sphere map is computed in the viewing plane. The width and height of the plane are equal to the diameter of the sphere. Rays fired using the orthographic projection are shown in blue (dark gray). In the center of the sphere, the

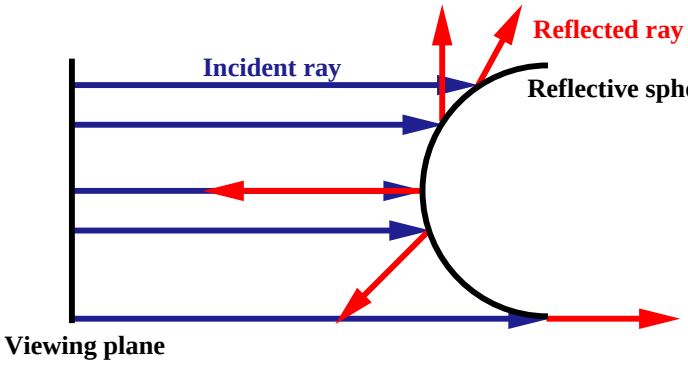


Figure 27. Creating a sphere map

ray reflects back to the viewer. Along the edges of the sphere, the rays are tangent and go behind the sphere.

Note that since the sphere map computes the irradiance at a single point, the sphere is infinitely small. Since the projection is orthographic, this implies that each texel in the image is also infinitely small. In effect, we take the limit as the size of the sphere (and the size of each texel) approaches 0. All of the rays along the outside of the sphere will map to the same point directly behind the sphere in the environment.

Using a Sphere Map OpenGL provides a mechanism to generate s and t texture coordinates at vertices based on the current normal and the direction to the eyepoint. The generated coordinates are then used to index a sphere map image which has been bound as a texture.

We denote the vector from the eye point to the vertex as u, normalized to u'. Since the computation is performed in eye coordinates, the eye is located at the origin and u is equal to the location of the vertex. The current normal n is transformed to eye coordinates, becoming n'. The reflected vector r can be computed as:

$$r = 2(n' \cdot u')n' - u' \tag{2}$$

We define:

$$m = 2\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2} \tag{3}$$

Then the texture coordinates are calculated as:

$$s = \frac{r_x}{m} + \frac{1}{2}$$

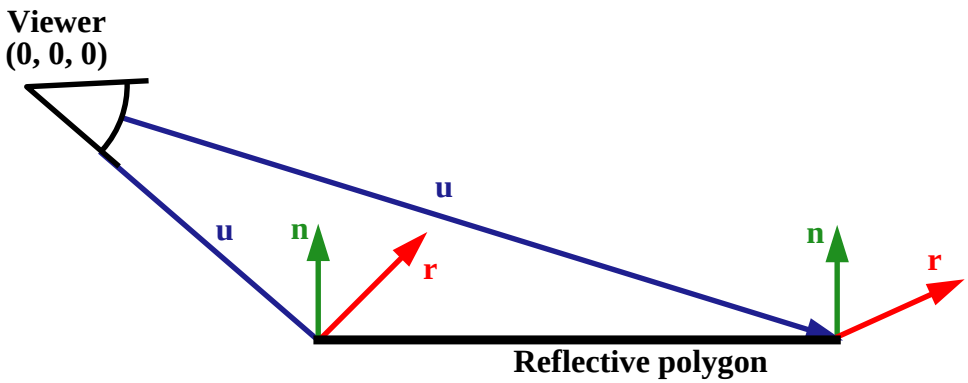


Figure 28. Sphere map coordinate generation

$$t = \frac{r_y}{m} + \frac{1}{2}$$

This computation happens internally to OpenGL in the texture coordinate generation step.

To use sphere mapping in OpenGL, the following steps are performed:

1. Bind the texture containing the sphere map
2. Set sphere mapping texture coordinate generation(`glTexGen(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP)`) and `glTexGen(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP)`)
3. Enable texture coordinate generation (`glEnable(TEXTURE_GEN_S)` and `glEnable(TEXTURE_GEN_T)`)
4. Draw the object, providing correct normals on a per-face or per-vertex basis

Generating a Sphere Map for Specular Reflection Several techniques exist to generate a specular sphere map. Two physical approaches are worth mentioning. In the first approach, the user literally takes a picture of a reflective sphere. Figure 29 was generated in this fashion. This technique is problematic in that the camera is visible in the reflection map. In the second approach, a fisheye lens approximates the sphere mapping. The problem with this technique is that no fisheye lens can provide the $360°$ field of view required for a correct result.

Figure 29. Reflection map created using a reflective sphere

A sphere map can also be generated programmatically. We consider the circle of the environment map within the square texture to be a unit circle. For each point (s, t) in the unit circle, we can compute a point p on the sphere:

$$\begin{aligned} p_x &= s \\ p_y &= t \\ p_z &= \sqrt{1.0 - p_x^2 - p_y^2} \end{aligned}$$

Since we are dealing with a unit sphere, the normal at p is equal to p. Given the vector e toward the eyepoint, we can compute the reflected vector r:

$$r = n * (n \cdot e) * 2 - e \tag{4}$$

In OpenGL, we assuming that the eyepoint is looking down the negative z axis, so $e = (0, 0, 1)$. Equation 4 reduces to:

$$\begin{aligned} r_x &= n_x * n_z * 2 \\ r_y &= n_y * n_z * 2 \\ r_z &= n_z * n_z * 2 - 1 \end{aligned}$$

The assumption that the $e = (0, 0, 1)$ means that OpenGL's sphere mapping is actually not view-independent. The implications of this assumption will be discussed below with the other limitations of the sphere mapping technique.

The rays are intersected with the environment to determine the irradiance. A simple implementation of the algorithm is shown in the following pseudocode:

```
void gen_sphere_map(GLsizei width, GLsizei height, GLfloat pos[3],
                    GLfloat (*tex)[3])
{
  GLfloat ray[3], color[3], p[3];
  GLfloat s,t;
  int i, j;

  for (j = 0; j < height; j++) {
    t = 2.0 * ((float)j / (float)(height-1) - .5);
    for (i = 0; i < width; i++) {
      s = 2.0 * ((float)i / (float)(width - 1) - .5);

      if (s*s + t*t > 1.0) continue;

      /* compute the point on the sphere (aka the normal) */
      p[0] = s;
      p[1] = t;
      p[2] = sqrt(1.0 - s*s - t*t);

      /* compute reflected ray */
      ray[0] = p[0] * p[2] * 2;
      ray[2] = p[1] * p[2] * 2;
      ray[3] = p[2] * p[2] * 2 - 1;
      fire_ray(pos, ray, tex[j*width + i]);
    }
  }
}
```

Note that we could easily optimize our routine such that the bounds on `i` in the inner `for` loop were intelligently set based on `j`.

We have encapsulated the most interesting part of the computation inside the `fire_ray` routine. `fire_ray` performs the ray/environment intersection given the starting point and the direction of the ray. Using the ray, it computes the color and puts the results into its third parameter (which is the appropriate location in the texture map).

A naive implementation such as the one above will lead to sampling artifacts. In reality, a texel in the image projects to a volume which should be intersected with the

environment. To filter, we should choose several rays in this volume and combine the results.

The intersection and color computation can be done in several ways. We may use a model of the scene and a ray tracing package. Alternately, we can represent the scene as six images which form the faces of a cube centered around the point for which the sphere map is being created. The images represent what a camera with a 90° field of view and a focal point at the center of the square would see in the given direction. The six images may be generated with OpenGL or a rendering package, or can be captured with a camera. Figure 30 shows six images which were acquired using a camera. Once the six images have been acquired, the rays from the point are intersected with the cube to provide the sphere map texel values. Figure 31 shows the map generated from the cube faces in Figure 30.

An alternate implementation uses OpenGL's texture mapping capabilities to create the sphere map. The algorithm takes as input the six cube faces. It then draws a tessellated hemisphere six times, mapping one of the faces into its correct location during each pass. The image of the sphere becomes the sphere map. Texture coordinates and the texture matrix combine to map the proper texels onto the sphere. At the vertices on the tessellated sphere, the values are correct. The interpolation between the vertices is not correct, but is generally a good approximation.

The texture mapping accelerated technique to generate sphere maps and the CPU technique described above are implemented in an example program found on the course web site.

Multipass Techniques and Interreflections Scenes containing two reflective objects may be rendered using sphere maps created via a multipass algorithm. We begin by creating an initial sphere map for each of the reflective objects in the scene. Choice of initial values was discussed in detail in Section 26. Then we iterate over the objects, recreating the sphere maps with the current sphere maps of the other objects applied. The following pseudocode illustrates how this algorithm might be implemented:

```
do {
  for (each reflective object obj with center c) {
    initialize the viewpoint to look along the axis (0, 0, -1)
    translate the viewpoint to c
    render the view of the scene (except for obj)
    save rendered image as cube1
    rotate the viewer to look along (0, 0, 1)
    render the view of the scene
    save rendered image as cube2
```

Figure 30. Image cube faces captured at a cafe in Palo Alto, CA

Figure 31. Sphere map generated from image cube faces in Figure 30

```
    rotate the viewer to look along (0, -1, 0)
    render the view of the scene
    save rendered image as cube3
    rotate the viewer to look along (0, 1, 0)
    render the view of the scene
    save rendered image as cube4
    rotate the viewer to look along (-1, 0, 0)
    render the view of the scene
    save rendered image as cube5
    rotate the viewer to look along (1, 0, 0)
    render the view of the scene
    save rendered image as cube6
    using the cube images, update the sphere map of obj
  }
} while (sphere map has not converged)
```

Note that during the rendering of the scene, other reflective objects must have their most recent sphere maps applied. Detection of convergence can be tricky. The simplest technique is to iterate a certain number of times and assume the results will be good. More sophisticated approaches can look at the change in the sphere maps for a given pass, or compute the maximum possible change given the projected area of the reflective objects. Once the sphere maps have been created we can draw the scene from any viewpoint. If none of the objects are moving, the sphere maps for each object can be created at program startup.

Other Sphere Mapping Techniques Sphere mapping may be used to approximate effects other the specular reflection. Any effect which is dependent only on the surface normal can be approximated, including Phong shading and refractive effects. We use our sphere map to store the outgoing color and intensity as a function of the normal. When computing our specular sphere map, this color was determined by firing a ray which had been reflected about the normal. To compute a different type of sphere map, we determine the color using a different method. For example, to create a Phong lighting map we can take the dot product of the normal direction and the direction to the light source.

Limitations of Sphere Mapping Although sphere mapping is generally convincing, it is not generally correct. Most of the artifacts come from the fact that the sphere map is generated at a single point and then applied over a large number of points. Objects with interreflections cannot be handled correctly. If reflected objects are close to the reflective object, their reflections should appear differently

when viewed from different points on the reflector. Using sphere maps, this will not happen. Sphere mapping results are only correct if we assume that all the reflective objects are infinitely far from the reflective object.

Fixing the eye point along the vector $(0, 0, 1)$ also leads to incorrect results. The same normal in eyespace will always map to the same location in the sphere map. A normal which points directly at the eyepoint maps to the center of the sphere map. A normal which points directly away from the user maps to the circle around the sphere map. Two important advantages of this simplification are that it significantly reduces the cost of computing r and that it ensures that the parts of the sphere map which have the best filtering are mapped to the primitives which face the user. In general, primitives which face the user will cover large areas in screen space and will be the focus of the user's attention.

Interpolation of the texture coordinates also leads to artifacts. Texture coordinates are computed at the vertices and linearly interpolated across the polygon. Unfortunately, the sphere map is not in a linear space, so this interpolation is not correct. Additionally, the linear interpolation will not take into account the fact that the points at the edge of the circle all map to the same location. Coordinates may be interpolated within the circle of the sphere map when they should be interpolated across the boundary.

9.4 Creating Shadows

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 area 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 with respect 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.

9.4.1 Projection Shadows

An easy-to-implement type of shadow can be created using projection transforms [46]. 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 z 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, the second time the object is rendered, the transform 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 polygon offset. This way the depth buffering will minimize the amount of clipping you'll have to do to the shadow.

The direction of the 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);
}
```

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.

9.4.2 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 [26].

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.

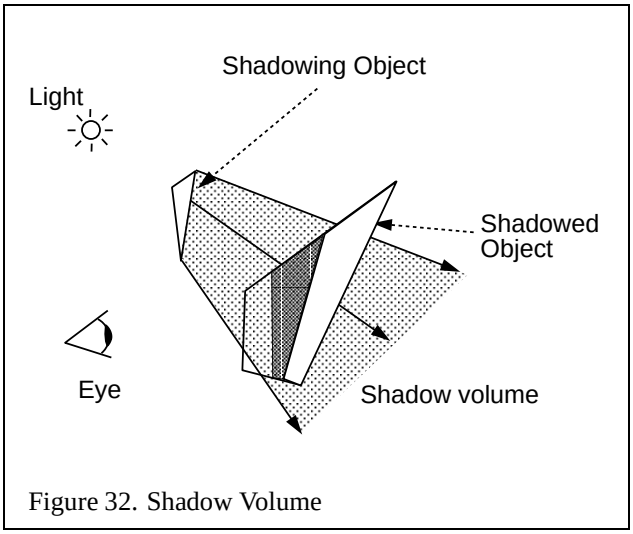


Figure 32. Shadow Volume

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 enabled.
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 was crossed an even number of times, 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 use only these for calculating the shadow volume. These vertices can be found by looking for any polygon edges that either (1) surround 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 which direction 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

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 light's 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, such as "double hits", the stencil count can get messed up, leading to grossly incorrect shadows.

9.4.3 Shadow Maps

Shadow maps use the depth buffer and projective texture mapping to create a screen space method for shadowing objects [39, 44]. Its 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 value, 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 1.1 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 shadowing object, or clip the object being shadowed. This is similar to the argument made for depth buffering vs. an object-based 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 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 are 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:

1. Increase shadow map/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 shadow map.

2. 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.

3. 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 to shadow itself. 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 shadow map technique to cast shadows from a light surrounded by objects. This is because the shadow map 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 the geometric relationship between the light and the objects in the scene.

9.4.4 Soft Shadows by Jittering Lights

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 modelled would be emitting energy. To reduce 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 with non-trivial patterns of light and dark, resulting from the light contributions of all the lights in the scene.

9.4.5 Soft Shadows Using Textures

Heckbert and Herf describe an alternative technique for rendering soft shadows by creating a texture for each partially shadowed polygon in the scene [24]. This texture represents the effect of the scene's lights on the polygon.

For each shadowed polygon, an image is rendered which represents the contribution of each light source for each shadowed polygon, and that image is used as a texture in the final scene containing the shadowed polygon. Shadowing polygons

are projected onto the shadowed polygon from the direction of the sample point on the light source. The accumulation buffer is used to average the results of that projection for several points (typically 16) on the polygon representing the light source.

The algorithm finds a single quadrilateral that tightly bounds the shadowed polygon in the plane of that polygon. The quad and the sample point on the light source are used to create a viewing frustum that projects intervening polygons onto the shadowed polygon. Multiple shadow textures per polygon are avoided because each "lighting" frustum shares the base quadrilateral, and so the shadowing results can all be accumulated into the same texture.

A pass is made for each sample point on each light source. The color buffer is cleared to the color of the light, and then the projected polygons are drawn with the ambient color of the scene. The resulting image is then added into the accumulation buffer. The final accumulation buffer result is copied into texture memory and is applied during the final scene as the polygon's texture.

Care must be taken to choose an image resolution for the shadow texture that looks acceptable on the final polygon. Depth testing and texturing can be disabled to improve performance during the projection pass. It may be necessary to save the accumulation buffer at intervals and average the results if the contribution of a shadow pass exceeds the resolution of the accumulation buffer.

A paper describing this technique in detail and other information on shadow generation algorithms is available at Heckbert and Herf's website [25].

10 Transparency

Transparent objects are common in everyday life and the addition of them can add significant realism to generated scenes. In this section, we will describe several techniques used to render transparent objects in OpenGL.

10.1 Screen-Door Transparency

One of the simpler transparency techniques is known as *screen-door transparency*. Screen-door transparency uses a bit mask to cause certain pixels not be rasterized. The percentage of bits in the bitmask which are 1 is equivalent to the transparency of the object.[14].

In OpenGL, screen-door transparency is implemented using *polygon stippling*. The command `glPolygonStipple` defines a 32x32 polygon stipple pattern. When stippling is enabled (using `glEnable(GL_POLYGON_STIPPLE)`) the low-order x and y bits of the screen coordinates of each fragment are used to index into the stipple pattern. If the corresponding bit of the stipple pattern is 0, the

fragment is rejected. If the bit is 1, rasterization continues.

Since the lookup into the stipple pattern takes place in screen space, a different pattern must be used for objects which overlap, even if the transparency of the objects is the same. Were the same stipple pattern to be used, the same pixels in the frame buffer would be drawn for each object. Of the transparent objects, only the last (or the closest, if depth buffering were enabled) would be visible.

The biggest advantage of screen-door transparency is that the objects do not need to be sorted. Also, rasterization may be faster on some systems using the screen-door technique than using other techniques such as alpha blending. Since the screen-door technique operates on a per-fragment basis, the results will not look as smooth as if another technique had been used.

10.2 Alpha Blending

To draw semi-transparent geometry, the most common technique is to use *alpha blending*. In this technique, the alpha value for each fragment drawn reflects the transparency of that object. Each fragment is combined with the values in the frame buffer using the blending equation

$$C_{out} = C_{src} * A_{src} + (1 - A_{src}) * C_{dst} \tag{5}$$

Here, C_{out} is the output color which will be written to the frame buffer. C_{src} and A_{src} are the source color and alpha, which come from the fragment. C_{dst} is the destination color, which is the color value currently in the frame buffer at the location. This equation is specified using the OpenGL command `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`. Blending is then enabled with `glEnable(GL_BLEND)`.

A common mistake when implementing alpha blending is to assume that it requires a frame buffer with an alpha channel. Note that the alpha values in the frame buffer (`GL_DST_ALPHA`) are not actually used, so no alpha buffer is required.

For the alpha blending technique to work correctly, the transparent primitives must be drawn in back to front order and must not intersect. To convince ourselves of this, we can consider two objects obj_1 and obj_2 with colors C_1 and C_2 and alphas A_1 and A_2. Assume that obj_2 is in front of obj_1 and that the frame buffer has been cleared to black. If obj_2 is drawn first, obj_1 will not be drawn at all unless depth buffering is disabled. Turning off depth buffering generally is a bad idea, but even if we could turn it off, the results would still be incorrect. After obj_2 had been drawn, the frame buffer color would be $C_2 * A_2$. After obj_1 had been drawn, the color would be $C_1 * A_1 + (1 - A_1) * C_2 * A_2$. If obj_1 had been drawn first, the value would be $C_2 * A_2 + (1 - A_2) * C_2 * A_2$. Sorting will be discussed in detail in Section 10.3.

The alpha channel of the fragment can be set in several ways. If lighting is not being used, then the alpha value can be set using a 4 component color command such as `glColor4fv`. If lighting is enabled, then the ambient and diffuse reflectance coefficients of the material should correspond to the translucency of the object.

If texturing is enabled, the source of the alpha channel is controlled by the texture internal format, the texture environment function, and the texture environment constant color. The interaction is described in more detail in the `glTexEnv` man page. Many intricate effects can be implemented using alpha values from textures.

10.3 Sorting

The sorting step can be 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, 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. Frequently only crude or perhaps no sorting at all gives acceptable results.

If there is a single transparent object, or multiple transparent objects which do not overlap in screen space (i.e. each screen pixel is touched by at most one of the transparent objects), a shortcut may be taken under certain conditions. If the objects are closed, convex, and viewed from the outside, culling may be used to draw the backfacing polygons prior to the front facing polygons. The steps are as follows:

1. Configure culling to eliminate front facing polygons: `glCullFace(FRONT)`
2. Enable backface culling: `glEnable(GL_CULL_FACE)`
3. Draw the object
4. Configure culling to eliminate backfacing polygons: `glCullFace(BACK)`
5. Draw the object again
6. Disable culling: `glDisable(GL_CULL_FACE)`

We assume that the vertices of the polygons of the object are arranged in a counterclockwise direction when the object is viewed from the outside. If necessary, we can specify that polygons oriented clockwise should be considered front-facing with the `glFrontFace` command.

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, 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.

10.4 Using the Alpha Function

The *alpha function* is used to discard fragments based upon a comparison of the fragment's alpha value with a reference value. The comparison function and the reference value are specified with the command `glAlphaFunc`. The alpha test is enabled with `glEnable(GL_ALPHA_TEST)`.

The alpha test is frequently used to draw complicated geometry using texture maps on polygons. For example, a tree can be drawn as a picture of a tree on a single rectangle. The parts of the texture which are part of the tree have an alpha value of 1; parts of the texture which are not part of the tree have an alpha value of 0. This technique is often combined with billboarding (Section 5.7), in which a rectangle is turned to perpetually face the eyepoint.

Like polygon stippling, the alpha function discards fragments instead of drawing them into the frame buffer. Therefore sorting of the primitives is not necessary (unless some other mode like alpha blending is enabled). The disadvantage is that pixels must be completely opaque or completely transparent.

10.5 Using Multisampling

On systems which support the multisample extension (`SGIS_multisample`), the per-fragment sample mask may be used to change the transparency of an object.

One technique involves `GL_SAMPLE_ALPHA_TO_MASK_SGIS`. If transparent objects in a scene do not overlap, `GL_SAMPLE_ALPHA_TO_MASK_SGIS` may be used. This parameter causes the alpha of a fragment to be mapped to a sample mask which will be bitwise anded with the fragment's mask. The value of the generated sample mask is implementation-dependent and is a function of the pixel location and the fragment's alpha value. If two objects were drawn at the same location with the same transparency, the sample mask would be the same and the same samples would be touched. If two objects were drawn at the same location with different transparencies, results may or may not be acceptable.

The simplest technique is to use the `glSampleMaskSGIS` command to set the value of the `GL_SAMPLE_MASK_SGIS`. This value is used to generate a temporary mask which is bitwise anded with the fragment's mask. Again, results may not be correct if transparent objects overlap.

Currently, `SGIS_multisample` is supported by Silicon Graphics and Hewlett Packard.

11 Natural Phenomena

The are a large number of naturally occurring phenomena such as smoke, fire and clouds which are challenging to render at interactive rates with any semblance of realism. A common solution is to reduce the requirement for complex geometry by using textures. Many of the techniques use a combination of geometry and texture which vary as a function of time or other parameters such as distance from the viewer.

11.1 Smoke

Modelling smoke potentially requires some sophisticated physics, but surprisingly realistic images can be generated using fairly simple techniques. One such technique involves capturing a 2D cross section or image of a puff of smoke with both luminance and alpha channels for the image. The image can then be texture mapped onto a quadrilateral and blended into the scene. The billboard techniques outlined in Section 5.7 can be used to ensure that the image is transformed to face the user. Using a `GL_MODULATE` texture environment, the color and alpha value of the quadrilateral can be used to control the color and transparency of the smoke in order to simulate different types of smoke. For example, smoke from an oil fire would be dark and opaque, whereas steam from a flare stack would be much lighter in color.

The size, position, orientation, and opacity of the quadrilateral can be varied as a function of time to simulate the puff of smoke enlarging, drifting and dissipating over time.

More realistic effects can be achieved using volumetric techniques. Instead of a 2D image, a 3D volumetric image of smoke is rendered using the algorithms described in Section 13. Again, dynamics can be simulated by varying the position, size and translucency of the volume. More complex dynamics can be simulated by applying local distortions or deformations to the texture coordinates of the volume lattice rather than simply applying uniform transformations. The volumetric shading technique described in Section 13.11 can be used to illuminate the smoke.

There are many procedural techniques which can be used to synthesize both 2D and 3D textures [13].

11.2 Vapor Trails

Vapor trails emanating from a jet or a missile can be rendered using methods similar to the painting technique described in Section 6.3. A circular, wispy 2D image such as that used in the preceding section is used to generate the vapor pattern over some unit interval by rendering it as a billboard. A texture image consisting only of alpha

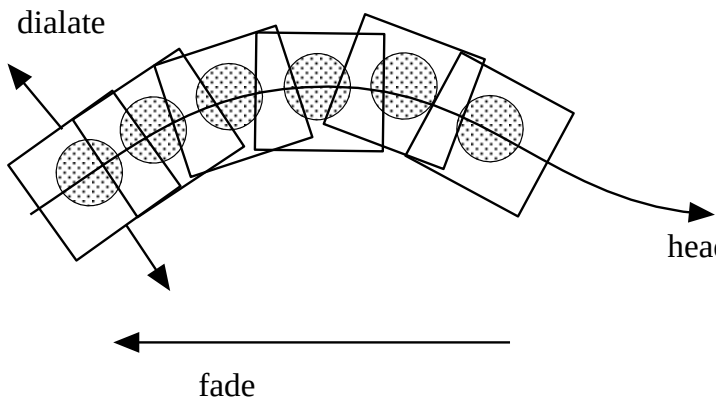


Figure 33. Vapor Trail

values is used to modulate the alpha values of a white billboard polygon. The trajectory of the airborne object is painted using multiple overlapping copies of the billboard as shown in Figure 33. Over time the individual billboards gradually enlarge and fade. The program for rendering a trail is largely an exercise in maintaining an active list of the position, orientation and time since creation for each billboard used to paint the trail. As each billboard polygon exceeds a threshold transparency value it can be discarded from the list.

11.3 Fire

The simplest techniques for rendering fire involve applying static images and movie loops as textures to billboards.

A static image of fire can be constructed from a noise texture; 16 describes how to make a noise texture using OpenGL. The weights for different frequency components should be chosen to reflect the spectral structure of fire, and turbulence can also be incorporated effectively into the texture. The texture is mapped to a billboard polygon. Several such textures, composited together, can create the appearance of multiple layers of intermingling flames. Finally, the texture coordinates may be distorted vertically to simulate the effect of flames rising and horizontally to mimic the effect of winds.

A sequence of fire textures can be played as an animation. The abrupt manner in which fire moves and changes intensity can be modelled using the same turbulence

techniques used to create the fire texture itself. The speed of the animation playback, as well as the distortion applied to the texture coordinates of the billboard, might be controlled using a turbulent noise function.

11.4 Clouds

Clouds, like smoke, have an amorphous structure without well defined surfaces and boundaries. In recent times, computationally intensive physical modelling techniques have given way to simplified mathematical models which are both computationally tractable and aesthetically pleasing [16, 13].

The main idea behind these techniques involves generating a realistic 2D or 3D texture function t using a fractal or spectral based function. Gardner suggests a Fourier-like sum of sine waves with phase shifts

$$t(x,y) = k \sum_{i=1}^{n} (c_i \sin(fx_i x + px_i) + t_0) \sum_{i=1}^{n} (c_i \sin(fy_i y + py_i) + t_0)$$

with the relationships

$$\begin{aligned} fx_{i+1} &= 2fx_i \\ fy_{i+1} &= 2fy_i \\ c_{i+1} &= .707c_i \\ px_i &= \frac{\pi}{2} \sin(fy_{i-1} y), i > 1 \\ py_i &= \frac{\pi}{2} \sin(fx_{i-1} x), i > 1 \end{aligned}$$

Care must be taken using this technique to choose values to avoid a regular pattern in the texture. Alternatively, texture generation techniques described in Section 16 can be used.

Either of these techniques can be used to produce a 2D texture which can be used to render a *cloud layer*. A cloud layer is simulated by drawing a large textured polygon in the sky at a fixed altitude. A luminance cloud texture can be blended with a blue polygon and a white constant texture environment color.

Some of the dynamic aspects of clouds can be simulated by vary parameters over time. Cloud development can be simulated by scaling and biasing the luminance values in the texture. Drifting can be simulated by moving the texture pattern across the sky, i.e., transforming the texture coordinates.

Gardner also suggests using ellipsoids to simulate 3D cloud structures. The texture data is generated using a 3-dimensional extension of the Fourier synthesis method outlined above and the textures are applied with increasing translucency near the boundary of the ellipsoid. These 3D textures can also be combined the volume rendering techniques described in Section 13 to produce 3D cloud images.

11.5 Water

A large body of research has been done into modelling, shading, and reproducing optical effects of water [49, 34, 15], yet most methods still present a large computation burden to achieve a realistic image. Nevertheless, it is possible to borrow from these approaches and achieve modest results while retaining interactive performance [28, 13].

The dynamics of wind and waves can be simulated using procedural models and rendered using meshes or height fields. The geometry is textured using simple procedural texture images. Multipass rendering techniques can be used to layer additional effects such as surf. Environment mapping can be used to simulate reflections from the surface. The combination of reflection mapping and a dynamic model for ripples provides a visually compelling image. Alternatively, synthetic perturbations to the texture coordinates as outlined in Section 5.13.7 can also be used.

Optical effects such as caustics can be approximated using parts of the OpenGL pipeline as described by Nishita and Nakamae [33] but interactive frame rates are not likely to be achieved. Instead such effects can be faked using textures to modulate the intensity of any geometry that lies below the surface.

11.6 Light Points

OpenGL has direct support for rendering both aliased and antialiased points, but these simple facilities are usually insufficient for simulating small light sources, such as stars, beacons, runway lights, etc. In particular, the size of OpenGL points is not affected by perspective projections. To render more realistic looking small light sources it is necessary to change some combination of the size and brightness of the source as a function of distance from the eye.

The brightness attenuation a as a function of distance, d, can be approximated by using the same equation used in the OpenGL lighting equation

$$\frac{1}{k_c + k_l d + k_q d^2}$$

Attenuation can be achieved by modulating the point size by the square root of the attenuation

$$size_{effective} = size \times \sqrt{a}$$

As the point size approaches the size of a single pixel the resolution of the raster display system will cause artifacts. To avoid this problem the point can be made

semi-transparent once it crosses a particular size threshold. The alpha value is proportional to the ratio of the point area determined from the size attenuation computation to the area of the point being rendered

$$alpha = \left(\frac{size_{effective}}{size_{threshold}} \right)^2$$

More complex behavior such as defocusing, perspective distortion and directionality of light sources can be achieved by using an image of the light lobe as a texture map combined with billboarding to keep the light lobe oriented towards the viewer.

11.7 Other Atmospheric Effects

OpenGL provides a primitive capability for rendering atmospheric effects such as fog, mist and haze. It is useful to simulate the affects of atmospheric effects on visibility to increase realism, and it allows the database designer to cover up a multitude of sins such as "dropping" polygons near the far clipping plane in order to sustain a fixed frame rate.

OpenGL implements fogging by blending the fog color with the incoming fragments using a fog blending factor, f,

$$C = fC_{in} + (1 - f)C_{fog}$$

This blending factor is computed using one of three equations: exponential (`GL_EXP`), exponential-squared (`GL_EXP2`), and linear (`GL_LINEAR`)

$$\begin{aligned} f &= e^{-(density \cdot z)} \\ f &= e^{-(density \cdot z)^2} \\ f &= \frac{end - z}{end - start} \end{aligned}$$

where z is the eye-coordinate distance between the viewpoint and the fragment center.

Linear fog is frequently used to implement intensity depth-cuing in which objects closer to the viewer are drawn at higher intensity [14]. The effect of intensity as a function of distance is achieved by blending the incoming fragments with a black fog color.

The exponential fog equation has some physical basis. It is the result of integrating a uniform attenuation between the object and the viewer. The exponential function can be used to represent a number of atmospheric effects using different

combinations of fog colors and density values. Since OpenGL does not fog the pixel values during a clear operation, the value of f at the far plane, far,

$$f_{far} = e^{-(density \cdot far)}$$

can be used to determine the color to which to clear the background

$$C_{bg} = f_{far}C_{in} + (1 - f_{far})C_{fog}$$

where C_{in} is the color to which the background would be cleared without fog enabled.

As mentioned earlier, the obscured visibility of objects near the far plane can be exploited to overcome various problems such as drawing time overruns, level-of-detail transitions, and database paging. However, in practice it has been found that the exponential function doesn't attenuate distant fragments rapidly enough, so exponential-squared fog was introduced to provide a sharper fall-off in visibility. Some vendors have gone a step further and provided more control over the fog function by allowing applications to control the fog value through a spline curve.

There are other problems that OpenGL's primitive fog model does not address. For example, emissive geometry such as the light points described above should be attenuated less severely than non-emissive geometry. This effect can be approximated by pre-compensating the color values for emissive geometry, or reducing the fog density when emissive geometry is drawn. Neither of these solutions is completely satisfactory since colors values are clamped 1.0 in OpenGL, limiting the amount of precompensation that can be done. Many OpenGL implementations use lookup table methods to efficiently compute the fog function, so changes to the fog density may result in expensive table recomputations. To overcome this problem some vendors have provided a mechanism to bias the eye-coordinate distance, avoiding the need to recompute the fog lookup table.

If OpenGL fog processing is bypassed it is possible to do more sophisticated atmospheric effects using multipass techniques. The OpenGL fog computation can be thought of as simple table lookup using the eye-coordinate distance. The result is used as a blend factor for blending between the fragment color and fog color. A similar operation can be implemented using `glTexGen` to generate the eye-coordinate distance for each fragment and a 1D texture for the fog function. Using a specially constructed 2D or 3D texture and a more sophisticated, texture coordinate generation function, it is possible to compute more complex fog functions incorporating parameters such as altitude and eye-coordinate distance.

12 Image Processing

12.1 Introduction

One of the strengths of OpenGL is that it provides tools for both image processing and 3D rendering. Unlike some libraries that contain only one or the other, OpenGL was designed with the understanding that many image processing tools are useful for 3D graphics. For example, convolution may be used to implement depth-of-field effects. Conversely, many operations typically thought of as image processing operations may be cast as geometric rendering and texture mapping operations. Electronic light tables (ELT's) used for defense imaging require image transformations which can be implemented using OpenGL's textured drawing capabilities. In this section, we will explore image processing applications of OpenGL, beginning with color manipulation, moving on to convolution, and finally discussing image warping. To solve these problems, we have three major parts of OpenGL at our disposal: the pixel transfer pipeline, geometric drawing and texturing, and fragment operations.

12.1.1 The Pixel Transfer Pipeline

The pixel transfer pipeline is the part of OpenGL most typically thought of in image processing applications. The pipeline is a configurable series of operations which are applied to each pixel during any command that moves pixels between the frame buffer, host memory, and texture memory, including:

- `glDrawPixels`
- `glReadPixels`
- `glTexImage2D`
- `glGetTexImage2D`
- `glCopyPixels`

These operations move image data which falls into one of the following categories:

- Color index values
- Stencil buffer values
- Depth values
- Color values (RGBA, luminance, luminance/alpha, red, green, ...)

The "pixel transfer pipeline" actually is four independent pipelines: one for each category of data.

For image processing, operations on color data are generally the most interesting. Before any operations are applied, source data in any color format (for example, `GL_LUMINANCE`) and type (for example, `GL_UNSIGNED_BYTE`) is converted into the canonical RGBA format, with each component represented as a floating-point value. All color pixel transfer operations are *defined* as operating on images of this type and format. After the pixel transfer operations have been applied, the image is converted to its destination type and format.

Base OpenGL defines only a few pixel transfer operations, which are controlled using the `glPixelTransfer` command. The operations are:

- `GL_INDEX_SHIFT` and `GL_INDEX_OFFSET`, which are applied only to color index images
- Scale and bias values which are applied to each channel of RGBA images
- Scale and bias values which are applied to depth values.
- Pixel maps, discussed in detail in Section 12.2.3

The pixel transfer pipeline is the part of OpenGL that has undergone the most growth through OpenGL extensions. Some of the more interesting extensions will be discussed in this section. We will list the vendors who have committed to support each extension as of April 1997. Where possible, we will mention techniques to achieve equivalent results on systems that do not support the extension.

12.1.2 Geometric Drawing and Texturing

OpenGL's texturing capabilities were discussed in detail in Section 5. These capabilities can be put to work to solve image processing problems. By texturing an input image onto a geometric grid, we can apply arbitrary deformations to the image. Given the textured draw rates of hardware-accelerated OpenGL platforms, very impressive performance can often be achieved though the use of textured geometric drawing. Image processing applications using texturing will be discussed in section 12.4.

12.1.3 The Frame Buffer and Per-Fragment Operations

Per-fragment and frame buffer operations can be used to perform operations on pixels of an image in parallel. Additionally, multiple images may be combined in a

variety of ways. Two main features are of interest: blending and the accumulation buffer. These features were discussed in detail in section 6. The accumulation buffer is particularly important since it provides several fundamental operations:

- Scaling of an image by a constant:
  - `glAccum(GL_MULT, <scale>)`
  - `glAccum(GL_LOAD, <scale>)`
  - `glAccum(GL_RETURN, <scale>)`
- Biasing of an image by a constant:
  - `glAccum(GL_ADD, <scale>)`
  - Clear of frame buffer with color `<scale>`, followed by `glAccum(GL_LOAD, 1)`
- Linear combination of two images on a pixel-by-pixel basis: `glAccum(GL_LOAD, <bias1>)` followed by `glAccum(GL_ACCUM, <bias2>)`

The accumulation buffer and blending will be discussed in subsequent sections in terms of the image processing operations they are used to implement.

12.2 Colors and Color Spaces

In this section we will consider ways to modify the pixels of an image on a local basis. That is, each output pixel will be a function of a single corresponding input pixel. Convolution, a non-local operation, will be considered in the next section.

12.2.1 The Accumulation Buffer: Interpolation and Extrapolation

Haeberli and Voorhies have suggested several interesting image processing techniques using linear interpolation and extrapolation. Each technique is stated in terms of the formula:

$$out = (1 - x) * in_0 + x * in_1 \tag{6}$$

The equation is evaluated on a per-pixel basis. in_0 and in_1 are the input images, out is the output image, and x is the blending factor. If x is between 0 and 1, the equations describe a linear interpolation. If x is allowed to range outside $[0..1]$, extrapolation results.[19]

In the limited case where $0 \leq x \leq 1$, these equations may be implemented using the accumulation buffer via the following steps:

1. Draw in_0 into the color buffer
2. Load in_0, scaling by $(1 - x)$ (`glAccum(GL_LOAD, (1-x))`)
3. Draw in_1 into the color buffer
4. Accumulate in_1, scaling by x (`glAccum(GL_ACCUM,x)`)
5. Return the results (`glAccum(GL_RETURN, 1)`)

We assume that in_0 and in_1 are between 0 and 1. Since the accumulation buffer can only store values in the range $[-1..1]$, for the case $x < 0$ or $x > 1$, the equation must be implemented in a different way. Given the value x, we can modify equation 6 and derive a list of accumulation buffer operations to perform the operation. We define a scale factor s such that:

$$s = max(|x|, |1 - x|)$$

Equation 6 becomes:

$$out = s(\frac{(1 - x)}{s} in_0 + \frac{x}{s} in_1)$$

and the list of steps becomes:

1. Compute s
2. Draw in_0 into the color buffer
3. Load in_0, scaling by $\frac{(1-x)}{s}$ (`glAccum(GL_LOAD, (1-x)/s)`)
4. Draw in_1 into the color buffer
5. Accumulate in_1, scaling by $\frac{x}{s}$ (`glAccum(GL_ACCUM, x/s)`)
6. Return the results, scaling by s (`glAccum(GL_RETURN, s)`)

The techniques suggested by Haeberli and Voorhies use a degenerate image as in_0 and an appropriate value of x to move toward or away from that image. To increase brightness, in_0 is set to a black image and $x > 1$. To change contrast, in_0 is set to a grey image of the average luminance value of in_1. Moving toward ($x < 1$) the grey image increases contrast; moving away decreases it. Saturation may be varied using a black and white version of in_1 as in_0 (for information on converting RGB images to luminance, see section 12.2.4). Sharpening may be accomplished by setting in_0 to a blurred version of in_1.[19] For more details, readers are encouraged to visit `http://www.sgi.com/grafica/interp/index.html`

12.2.2 Pixel Scale and Bias Operations

Scale and bias operations can be used to adjust the colors of images. Also, they can be used to select and expand a small range of values in the input image. Scales and biases are applied at several locations in the pixel transfer pipeline. In general, scales and biases are controlled with eight floating point values (a scale and a bias for each channel).

The first scale and bias in the pixel transfer pipeline is part of base OpenGL and is specified with `glPixelTransfer(<pname>, <value>)` where `<pname>` specifies one of `GL_RED_SCALE`, `GL_RED_BIAS`, `GL_GREEN_SCALE`, `GL_GREEN_BIAS`, `GL_BLUE_SCALE`, `GL_BLUE_BIAS`, `GL_ALPHA_SCALE`, or `GL_ALPHA_BIAS`. Other scale and bias steps are associated with the color matrix extension (`SGI_color_matrix`) and the convolution extension (`EXT_convolution`).

12.2.3 Look-Up Tables

One useful tool for color modification is the look-up table. Generally speaking, a look-up table takes a value, maps it to a location in a table, and replaces the incoming value with the contents of the table entry. OpenGL provides three mechanisms which are basically look-up tables. Two, pixel maps and color tables, look up components independently in one-dimensional tables. These mechanisms provide efficient mapping for applications requiring no between the channels of the image. A third mechanism, pixel texturing, uses the OpenGL texturing capability to perform multi-dimensional look-ups.

Pixel Maps *Pixel maps* are a feature of base OpenGL which allow certain look-up operations to be performed. OpenGL maintains tables which map:

- The red channel to the red channel (`GL_PIXEL_MAP_R_TO_R`)
- The green channel to the green channel (`GL_PIXEL_MAP_G_TO_G`)
- The blue channel to the blue channel (`GL_PIXEL_MAP_B_TO_B`)
- The alpha channel to the alpha channel (`GL_PIXEL_MAP_A_TO_A`)
- Color indices to color indices (`GL_PIXEL_MAP_I_TO_I`)
- Stencil indices to stencil indices (`GL_PIXEL_MAP_S_TO_S`)
- Color indices to RGBA values (`GL_PIXEL_MAP_I_TO_R`, `GL_PIXEL_MAP_I_TO_G`, `GL_PIXEL_MAP_I_TO_B`, and `GL_PIXEL_MAP_I_TO_A`)

Tables that map color indices to RGBA values are used automatically whenever an image with a color index format is transferred to a destination which requires an RGBA image. For example, performing a `glDrawPixels` of a color index image to an RGBA frame buffer would result in application of the I to RGBA pixel maps. Other tables are enabled with the commands `glPixelTransfer(GL_MAP_COLOR, 1)` and `glPixelTransfer(GL_MAP_STENCIL, 1)`.

Pixel maps are defined using the `glPixelMap` command and queried using the `glGetPixelMap` command. Details on the use of these commands may be found in [7]. The sizes of the pixel maps are not tied together in any way. For example, the R to R pixel map does not need to be the same size as the G to G pixel map.

Each system provides a constant, `GL_MAX_PIXEL_MAP_TABLE`, which gives the maximum size of a pixel map which may be defined.

The Color Table Extension The *color table extension*, `SGI_color_table`, provides additional look-up tables in the OpenGL pixel transfer pipeline. Although the capabilities of color tables and pixel maps are similar, the semantics are different.

The color table extension defines the following look-up tables:

- "First" color table (`GL_COLOR_TABLE_SGI`)
- Post convolution color table (`GL_POST_CONVOLUTION_COLOR_TABLE_SGI`)
- Post color matrix color table (`GL_POST_COLOR_MATRIX_COLOR_TABLE_SGI`)

Each table is independently enabled and disabled using the `glEnable` and `glDisable` commands. One, two, or all three of the tables may be applied during the same operation. Color tables do not operate on color index images, unless the color index image was previously converted to an RGBA image by the I to RGBA pixel maps as described in the previous section.

Color tables are specified using the `glColorTableEXT` and `glCopyColorTableEXT` commands and are queried using the `glGetColorTableEXT` command. The man pages for these commands provide details on their use. Note that unlike the RGBA to RGBA pixel maps, all channels of a color table are specified at the same time.

When a color table is specified, an internal format parameter (for example, `GL_RGB` or `GL_LUMINANCE_EXT`) gives the channels present in the table. When the color table is applied to an image (which is by definition RGBA), channels of the image which are not present in the color table are left unmodified. In this way,

color tables are more flexible than pixel mpas, which map and replace all channels of the input image.

Although color tables provide a similar functionality to pixel maps and may prove more useful in certain circumstances, they do not replace pixel maps in the OpenGL pipeline and the tables managed by pixel maps and color tables are independent. It is possible to apply both a pixel map and a color table (or color tables) during the same pixel operation (although the utility of this is questionable). The maximum sizes and relative efficiencies of pixel maps and color tables vary from platform to platform.

The color table extension is currently supported by the following vendors:

- Silicon Graphics
- Hewlett Packard
- Sun Microsystems, Inc.

The Texture Color Table Extension The texture color table extension (`SGI_texture_color_table`) provides a color table (`GL_TEXTURE_COLOR_TABLE_SGI`) which is applied to texels after filtering and prior to combination with the fragment color with the texture environment operation. The procedures to define, enable, and disable the texture color table are the same as those of the tables in `SGI_color_table`.

The texture color table extension is currently supported by the following vendors:

- Silicon Graphics
- Evans & Sutherland
- Hewlett Packard
- Sun Microsystems, Inc.

The Pixel Texture Extension The pixel texture extension (`SGIX_pixel_texture`) allows multi-dimensional lookups through OpenGL's texturing capability. Remember that OpenGL defines rasterization of a pixel image during a `glDrawPixels` or `glCopyPixels` command as the generation of a fragment for each pixel in the image. Per-fragment operations are applied, including texturing (if enabled). If the input image contained color data, each fragment's color comes from the color of the pixel that generated it. The texture coordinate of the fragment is taken from the current raster position, which is generally not useful

because the texture coordinate will be constant over the pixel rectangle. The pixel texture extension allows the texture coordinates s, t, q, and r of the fragment to be copied from the color coordinates R, G, B, and A of the pixel. With three and four dimensional textures (`EXT_texture3D` and `SGIS_texture4D`), arbitrary effects can be implemented (although the texture storage requirements to do so can be staggering).

The pixel texture extension is supported by the following vendors:

- Silicon Graphics

Equivalent Functionality Without `SGIX_pixel_texture` There is no way to apply a true multidimensional lookup to a pixel image without `SGIX_pixel_texture`. In some cases, pixel maps and color tables may be used as a substitute. Blending, accumulation buffer operations, or scale/bias operations may be used when the function to be applied is linear and each channel is independent. In other cases, the application will have to perform the lookup on the host or draw a textured point for each pixel in the image.

12.2.4 The Color Matrix Extension

The color matrix extension (`SGI_color_matrix`) defines a 4x4 color matrix which is managed using the same commands as the projection, modelview, or texture matrix. The color matrix premultiplies RGBA colors in the pixel transfer pipeline and as such can be used to perform linear color space conversions.

Since the color matrix is treated like any other matrix, it is always enabled and defaults to the identity. To change the contents of the color matrix, the current matrix mode must be set to `GL_COLOR` using the `glMatrixMode` command. After that, the color matrix may be manipulated using the same commands as any other matrix. The commands `glLoadMatrix`, `glPushMatrix`, and `glPopMatrix` generally prove the most useful.

The color matrix extension is currently supported on the following platforms:

- Silicon Graphics

Equivalent Functionality Without `SGI_color_matrix` Unfortunately, the functionality of `SGI_color_matrix` is difficult to efficiently duplicate on systems which do not support the extension. In the case where the image is going from the host to the framebuffer (a `glDrawPixels` operation), the best way to handle the situation is the split the image up into red, green, blue, and alpha images (via application processing or a draw followed by reads with `format` set

to GL_RED, GL_GREEN, GL_BLUE, or GL_ALPHA). The red, green, blue, and alpha images can be drawn as GL_LUMINANCE images. RGBA scale operations are applied, with the four values equal to the row of the matrix corresponding to source channel. The images are composited in the frame buffer using blending (glBlendFunc(GL_ONE, GL_ONE)).

Scale and Bias Scale and bias operations may be performed using the color matrix. A scale factor can be applied using the glScale command. A bias is equivalent to a translation and may be applied using the glTranslate command. Using glScale and glTranslate, the R scale or bias is put in the x parameter, the G scale or bias in the y parameter, and the B scale or bias in the z parameter. Modifications to the A channel must be specified using glLoadMatrix or glMultMatrix. In general using the color matrix will be slower than using a transfer operation which implements scale and bias directly, but management of state may be easier using color matrices. Also, the scale and bias could be rolled into another color matrix operation.

Conversion to Luminance Converting a color image into a luminance image may be accomplished by putting the weights for R, G, and B along the top row of the matrix:

$$\begin{bmatrix} L \\ L \\ L \\ 0 \end{bmatrix} = \begin{bmatrix} R_w & G_w & B_w & 0 \\ R_w & G_w & B_w & 0 \\ R_w & G_w & B_w & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \\ A \end{bmatrix}$$

The recommended weight values for R_w, G_w, and B_w are 0.3086, 0.6094, and 0.0820. Some authors have used the values from the YIQ color conversion equation (0.299, 0.587, and 0.114), but Haeberli notes that these values are incorrect in a linear RGB color space.[18]

Modifying Saturation The *saturation* of a color is the distance of that color from a grey of equal intensity.[14] Haeberli has suggested modifying saturation using the equation:

$$\begin{bmatrix} R' \\ G' \\ B' \\ A \end{bmatrix} = \begin{bmatrix} a & d & g & 0 \\ b & e & h & 0 \\ c & f & i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \\ A \end{bmatrix}$$

where:

$$a = (1 - s) * R_w + s$$

$$
\begin{aligned}
b &= (1-s) * R_w \\
c &= (1-s) * R_w \\
d &= (1-s) * G_w \\
e &= (1-s) * G_w + s \\
f &= (1-s) * G_w \\
g &= (1-s) * B_w \\
h &= (1-s) * B_w \\
i &= (1-s) * B_w + s
\end{aligned}
$$

with R_w, G_w, and B_w as described in the above section. Since the saturation of a color is the difference between the color and a grey value of equal intensity, it is comforting to note that setting s to 0 gives the luminance equation. Setting s to 1 leaves the saturation unchanged; setting it to -1 takes the complement of the colors.[18]

Hue Rotation Changing the hue of a color may be accomplished by loading a rotation about the grey vector $(1, 1, 1)$. This operation may be performed in one step using the `glRotate` command. The matrix may also be constructed via the following steps:[18]

1. Load the identity matrix (`glLoadIdentity`)
2. Rotate such that the grey vector maps onto the Z axis using the `glRotate` command
3. Rotate about the Z axis to adjust the hue (`glRotate(<degrees>, 0, 0, 1)`)
4. Rotate the grey vector back into position

Unfortunately, a naive application of `glRotate` will not preserve the luminance of the image. To avoid this problem, we must make sure that areas of constant luminance map to planes perpendicular to the Z axis when we perform the hue rotation. Recalling that the luminance of a vector (R, G, B) is equal to:

$$(R, G, B) \cdot (R_w, G_w, B_w)$$

we realize the a plane of constant luminance k is defined by:

$$(R, G, B) \cdot (R_w, G_w, B_w) = k$$

Therefore, the vector (R_w, G_w, B_w) is perpendicular to planes of constant luminance. The algorithm for matrix construction becomes the following:[18]

1. Load the identity matrix
2. Apply a rotation matrix M such that the grey vector $(1, 1, 1)$ maps onto the positive Z axis
3. Compute $(R'_w, G'_w, B'_w) = M(R_w, G_w, B_w)$ Apply a skew transform which maps (R'_w, G'_w, B'_w) to $(0, 0, B'_w)$. This matrix is:

$$\begin{bmatrix} 1 & 0 & \frac{-R'_w}{B'_w} & 0 \\ 0 & 1 & \frac{-G'_w}{B'_w} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

4. Rotate about the Z axis to adjust the hue
5. Apply the inverse of the shear matrix
6. Apply the inverse of the rotation matrix

It is possible to compute a single matrix as a function of R_w, G_w, B_w, and the degrees of rotation which would perform the operation.

CMY Conversion The CMY color space describes colors in terms of the subtractive primaries: cyan, magenta, and yellow. CMY is used mainly for hardcopy devices such as color printers. Generally, the conversion from RGB to CMY follows the equation:[14]

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

CMY conversion may be performed using the color matrix or a scale and bias operation. The conversion is equivalent to a scale by -1 and a bias by $+1$. Using the 4x4 color matrix, the equation may be restated as:

$$\begin{bmatrix} C \\ M \\ Y \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 1 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \\ 1 \end{bmatrix}$$

Here, we require that the incoming alpha channel be equal to 1. If the source is RGB, the 1 will be added automatically in the format conversion stage of the pipeline.

A related color space, CMYK, uses a fourth channel (K) to represent black. Since conversion to CMYK requires a $min()$ operation, it cannot be performed using the color matrix.

An extension, `CMYKA`, also supports conversion to and from CMYK and CMYKA. This extension is currently supported by Evans & Sutherland.

YIQ Conversion The YIQ color space is used in U.S. color television broadcasting. Conversion from RGBA to YIQA may be accomplished using the color matrix:

$$\begin{bmatrix} Y \\ I \\ Q \\ A \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 & 0 \\ 0.596 & -0.275 & -0.321 & 0 \\ 0.212 & -0.523 & 0.311 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \\ A \end{bmatrix}$$

(Generally, YIQ is not used with an alpha channel so the fourth component is eliminated.) The inverse matrix is used to map YIQ to RGBA.[14]

12.3 Convolutions

12.3.1 Introduction

Convolutions are used to perform many common image processing operations including sharpening, blurring, noise reduction, embossing, and edge enhancement. In this section, we begin with a very brief overview of the mathematics of the convolution operation. More detailed explanations of the mathematics and uses of the convolution operation can be found in many books on computer graphics and image processing. One good reference is [14]. After our brief mathematical introduction, we will describe two ways to perform convolutions using OpenGL: via the accumulation buffer and via the convolution extension.

12.3.2 The Convolution Operation

The *convolution operation* is a mathematical operation which takes two functions $f(x)$ and $g(x)$ and produces a third function $h(x)$. Mathematically, convolution is defined as:

$$h(x) = f(x) * g(x) = \int_{-\infty}^{+\infty} f(\tau) g(x - \tau) d\tau \tag{7}$$

$g(x)$ is referred to as the *filter*. The integral only needs to be evaluated over the range where $g(x - \tau)$ is nonzero (called the *support* of the filter).[14]

In spatial domain image processing, we discretize the convolution operation. $f(x)$ becomes an array of pixels $F[x]$. The kernel $g(x)$ is an array of values

$G[0...(width-1)]$ (we assume finite support). Equation 7 becomes:

$$H[x] = \sum_{i=0}^{width-1} F[x+i]G[i] \tag{8}$$

Two-Dimensional Convolutions Since in image processing we generally operate on two-dimensional images, we extend equation 8 to:

$$H[x][y] = \sum_{j=0}^{height-1} \sum_{i=0}^{width-1} F[x+i][y+j]G[i][j] \tag{9}$$

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 the kernel, 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.

Separable Filters In the general case, the two-dimensional convolution operation requires $(width * height)$ multiplications for each output pixel. Separable filters are a special case of general convolution in which the filter

$$G[0..(width-1)][0..(height-1)]$$

can be expressed in terms of two vectors

$$G_{row}[0..(width-1)]G_{col}[0..(height-1)]$$

such that for each $(i,j)\epsilon([0..(width-1)],[0..(height-1)])$

$$G[i][j] = G_{row}[i] * G_{col}[j]$$

If the filter is separable, the convolution operation may be performed using only $(width + height)$ multiplications for each output pixel. Equation 9 becomes:

$$H[x][y] = \sum_{j=0}^{height-1} \sum_{i=0}^{width-1} F[x+i][y+j]G[i][j] =$$

$$\sum_{j=0}^{height-1} \sum_{i=0}^{width-1} F[x+i][y+j]G_{row}[i]G_{col}[j] =$$

$$\sum_{j=0}^{height-1} G_{col}[j] \sum_{i=0}^{width-1} F[x+i][y+j]G_{row}[i]$$

To apply the separable convolution, we first apply G_{row} as though it were a $width$ by 1 filter. We then apply G_{col} as though it were a 1 by $height$ filter.

12.3.3 Convolutions Using the Accumulation Buffer

The convolution operation may be implemented using the accumulation buffer. The input image is stored in the color buffer and read by the `glAccum` function. The output image is built up in the accumulation buffer. For each kernel entry $G[i][j]$, we translate the input image by $(-i,-j)$ from its original position. The translation may be accomplished using the `glCopyPixels` command. We then accumulate the translated image using the command `glAccum(GL_ACCUM, G[i][j])`. $width * height$ translations and accumulations must be performed.

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 is defined as:

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

Since the accumulation buffer can only store values in the range (-1..1), we first modify the kernel such that at any point in the computation the values do not exceed this range. Assuming the input images values are in the range (0..1), the modified kernel is:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} = 4 * \begin{bmatrix} \frac{-1}{4} & \frac{-2}{4} & \frac{-1}{4} \\ 0 & 0 & 0 \\ \frac{1}{4} & \frac{2}{4} & \frac{1}{4} \end{bmatrix}$$

The operations needed to apply the filter are:

1. Draw the input image
2. `glAccum(GL_LOAD, 1/4)`
3. Translate the input image left by one pixel
4. `glAccum(GL_ACCUM, 2/4)`
5. Translate the input image left by one pixel
6. `glAccum(GL_ACCUM, 1/4)`
7. Translate the input image right by two pixels and down by two pixels
8. `glAccum(GL_ACCUM, -1/4)`
9. Translate the input image left by one pixel
10. `glAccum(GL_ACCUM, -2/4)`
11. Translate the input image left by one pixel
12. `glAccum(GL_ACCUM, -1/4)`
13. Return the results to the frame buffer (`glAccum(GL_RETURN, 4)`)

In this example, each pixel in the output image is the combination of pixels in the 3 by 3 pixel square whose lower left corner is at the output pixel. 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. As an optimization, we ignore locations where the kernel is equal to zero.

A general algorithm for the 2D convolution operation is:

```
Draw the input image
for (j = 0; j < height; j++) {
  for (i = 0; i < width; i++) {
    glAccum(GL_ACCUM, G[i][j]*scale);
    Move the input image to the left by 1 pixel
  }
  Move the input image to the right by width pixels
  Move the input image down by 1 pixel
}
glAccum(GL_RETURN, 1/scale);
```

`scale` is a value chosen to ensure that the intermediate results cannot go outside a certain range. In the Sobel filter example, `scale = 4`. Assuming the input values are in $(0..1)$, `scale` can be naively computed using the following algorithm:

```
float minPossible = 0, maxPossible = 1;
for (j = 0; j < height; j++) {
  for (i = 0; i < width; i++) {
    if (G[i][j] < 0) {
       minPossible += G[i][j];
    } else {
       maxPossible += G[i][j];
    }
  }
}
scale = 1.0 / ((-minPossible > maxPossible) ?
               -minPossible : maxPossible);
```

Since the accumulation buffer has limited precision, more accurate results could be obtained by changing the order of the computation and computing `scale` accordingly. Additionally, if the input image can be constrained to a smaller range, `scale` can be made larger, which may also give more accurate results.

For separable kernels, convolution can be implemented using $width + height$ image translations and accumulations. A general algorithm is:

```
Draw the input image
for (i = 0; i < width; i++) {
  glAccum(GL_ACCUM, Grow[i]);
  Move the input image to the left 1 pixel
}
glAccum(GL_RETURN, 1);
for (j = 0; j < height; j++) {
  glAccum(GL_ACCUM, Gcol[j]);
  Move the frame buffer image down by 1 pixel
}
glAccum(GL_RETURN, 1);
```

In this example, we have assumed that the row and column filters have been constructed such that the accumulation buffer values will never go out of range. For the general case, a scale value may be needed. More accurate results may be obtained if scale values are computed independently for the row and column steps. An accumulation buffer multiply in between the two steps may be required.

12.3.4 The Convolution Extension

The *convolution extension*, `EXT_convolution`, defines a stage in the OpenGL pixel transfer pipeline which applies a 1D, separable 2D, or general 2D convolution. The 1D convolution is applied only to 1D texture downloads and is infrequently used. 2D kernels are specified using the commands `glConvolutionFilter2DEXT`, `glCopyConvolutionFilter2DEXT`, and `glSeparableFilter2DEXT`. The convolution stage is enabled using `glEnable`. Filters are queried using `glGetConvolutionFilterEXT` and `glGetSeparableFilterEXT`.

The maximum permitted convolution size is machine-dependent and may be queried using `glGetConvolutionParameterfvEXT` with the parameters `GL_MAX_CONVOLUTION_WIDTH_EXT` and `GL_MAX_CONVOLUTION_HEIGHT_EXT`.

The relative performance of separable and general filters varies from platform to platform, but it is best to specify a separable filter whenever possible.

`EXT_convolution` is currently supported by the following vendors:

- Silicon Graphics
- Hewlett Packard
- Sun Microsystems, Inc.

12.3.5 Useful Convolution Filters

In this section, we briefly describe several useful convolution filters. The filters may be applied to an image using either the convolution extension or the accumulation buffer technique. Unless otherwise noted, the kernels presented are normalized (that is, the kernel weights sum to 0).

The reader should keep in mind that this section is intended only as a very basic reference. Numerous texts on image processing provide more details and other filters. All information presented in this section comes from [31].

Line detection Detection of one pixel wide lines can accomplished with the following filters:

Horizontal Edges

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

Vertical Edges

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

Left Diagonal Edges

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

Right Diagonal Edges

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

Gradient Detection (Embossing) Changes in value over 3 pixels can be detected using kernels called *Gradient Masks* or *Prewitt Masks*. The direction of the change from darker to lighter is described by one of the points of the compass. The 3x3 kernels are as follows:

North

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

West

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

East

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

South

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

Northeast

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

Smoothing and Blurring Smoothing and blurring operations are low-pass spatial filters. They reduce or eliminate high-frequency aspects of an image.

Arithmetic Mean The arithmetic mean simply takes an average of the pixels in the kernel. Each element in the filter is equal to 1 divided by the total number of elements in the filter. Thus the 3x3 arithmetic mean filter is:

$$\begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}$$

Basic Smooth: 3x3 (not normalized)

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Basic Smooth: 5x5 (not normalized)

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 4 & 4 & 4 & 1 \\ 1 & 4 & 12 & 4 & 1 \\ 1 & 4 & 4 & 4 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

High-pass Filters A high-pass filter enhances the high-frequency parts of an image. This type of filter is used to sharpen images.

Basic High-Pass Filter: 3x3

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Basic High-Pass Filter: 5x5

$$\begin{bmatrix} 0 & -1 & -1 & -1 & 0 \\ -1 & 2 & -4 & 2 & -1 \\ -1 & -4 & 13 & -4 & -1 \\ -1 & 2 & -4 & 2 & -1 \\ 0 & -1 & -1 & -1 & 0 \end{bmatrix}$$

Laplacian Filter The *Laplacian* is used to enhance discontinuities. The 3x3 kernel is:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

and the 5x5 is:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 24 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Sobel Filter The *Sobel filter* consists of two kernels which detect horizontal and vertical changes in an image. If both are applied to an image, the results can by used to compute the magnitude and direction of the edges in the image. If the application of the Sobel kernels results in two images which are stored in the arrays `Gh[0..(height-1][0..(width-1)]` and `Gv[0..(height-1)][0..(width-1)]`, the magnitude of the edge passing through the pixel `x, y` is given by:

$$M_{sobel}[x][y] = \sqrt{Gh[x][y]^2 + Gv[x][y]^2} = |Gh[x][y]| + |Gv[x][y]|$$

(we are justified in using the magnitude representation since the values represent the magnitude of orthogonal vectors). The direction can also be derived from `Gh` and `Gv`:

$$\phi_{sobel}[x][y] = tan^{-1}(\frac{Gv[x][y]}{Gh[x][y]})$$

The 3x3 Sobel kernels are:

Horizontal

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

Vertical

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

12.4 Image Warping

12.4.1 The Pixel Zoom Operation

OpenGL provides control over the generation of fragments from pixels via the pixel zoom operation. Zoom factors are specified using `glPixelZoom`. Negative zooms are used to specify reflections.

Pixel zooming may prove faster than the texture mapping techniques described below on some systems, but do not provide as fine a control over filtering.

12.4.2 Warps Using Texture Mapping

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.

Line Integral Convolution Brian Cabral and Casey Leedom have developed a technique for vector field visualization known as *line integral convolution*.[8] The technique takes an input vector field and an input image. For each location p in the input vector field, a parametric curve $P(p, s)$ is generated which passes through the location and follows the vector field for some distance in either direction. To create an output pixel $F'(p)$, a weighted sum of the values of the input image F along the curve is computed. The weighting function is $k(x)$. Thus the continuous form of the equation is:

$$F'(p) = \frac{\int_{-L}^{L} F(P(p,s))k(s)ds}{\int_{-L}^{L} k(s)ds}$$

To discretize the equation, we use values $P_{0..l}$ along the curve $P(p, s)$:

$$F'(p) = \frac{\sum_{i=0}^{l} F(P_i)h_i}{\sum_{i=0}^{l} h_i}$$

The computation of the output values of this equation may accelerated using OpenGL. We use a mesh texture mapped with the input image to create the output image. The mesh is redrawn l times. At each step, we advect the texture coordinates and accumulate the results. *Advection* applies a mapping defined by the vector field to the input points. A simple advection implementation moves each point by a fixed amount in the direction of the vector flow at the point. Advection has been well-studied, and many more complicated algorithms exist.

An implementation of the algorithm uses the following variables:

- `int l`: Number of steps
- `GLfloat h[0..(l-1)]`: Kernel weights
- `GLfloat hNormalize`: Normalization factor ($\sum_{i=0}^{l} h_i$)
- `GLfloat gridW, gridH`: Size of the grid
- `GLfloat *grid[2]`: Grid of texture coordinates.

and the functions:

- `advect_grid(GLfloat s)`: Advect grid by `s`, which may be positive or negative.

We begin by initializing the grid points:

```
void init(void)
{
  int x, y;

  for (y = 0; y < gridH; y++) {
    for (x = 0; x < gridW; x++) {
      grid[y*gridW + x][0] = x;
      grid[y*gridW + x][1] = y;
    }
  }
}
```

The texture image is then downloaded and bound. In the draw routine we call:

```
void lic(void)
{
  int x, y;
  int i;

  advect_grid(-l/2);

  glClear(GL_COLOR_BUFFER_BIT | GL_ACCUM_BUFFER_BIT);

  /* scale texture coordinates */
  glPushAttrib(GL_TRANSFORM_BIT);
  glMatrixMode(GL_TEXTURE);
  glPushMatrix();
  glScalef(1.0/(gridW-1), 1.0/(gridH-1), 1);

  for (i = 0; i < l; i++) {
    glEnable(GL_TEXTURE_2D);
    for (y = 0; y < gridH-1; y++) {
      glBegin(GL_QUAD_STRIP);
      for (x = 0; x < gridW-1; x++) {
        glTexCoord2fv(grid[y*gridW + x]);
        glVertex2i(x, y);
        glTexCoord2fv(grid[y*gridW + x+1]);
        glVertex2i(x+1, y);
        glTexCoord2fv(grid[(y+1)*gridW + x]);
        glVertex2i(x, y+1);
        glTexCoord2fv(grid[(y+1)*gridW + x+1]);
        glVertex2i(x+1, y+1);
      }
      glEnd();
    }
    glDisable(GL_TEXTURE_2D);
    glAccum(GL_ACCUM, h[i]);

    advect_grid(1);
  }

  glAccum(GL_RETURN, hNormalize);

  glPopMatrix();
```

```
  glPopAttrib();
}
```

In the `lic` routine, we first clear the color and accumulation buffers. Next, we modify the texture matrix such that a texture coordinate of `(gridW, gridH)` will map to the upper right corner of the input texture.

Upon each iteration of the loop, we draw the grid using the array of texture coordinates (vertex arrays could provide a more efficient implementation). Then, we accumulate the results, weighting by the kernel array entry. Next, we call `advect_grid` to update the texture coordinate array. At the end of the routine, we return the results and normalize by the sum of the kernel weights.

Upon implementation, several difficulties may present themselves. First, implementing `advect_grid` well is non-trivial (but well-studied). Second, here we have used a static grid to draw the field. This approach will probably lead to artifacts when drawing high-frequency fields or unnecessary inefficiency when drawing low-frequency fields. A better approach would subdivide the grid based on the behavior of the vector field. Also, the user may find that the results of the accumulation operation go outside the range $[-1..1]$ if care is not taken when choosing the kernel and normalization values. Finally, dealing with the three different coordinate spaces (vector field, grid, and texture image) can become complicated.

13 Volume Visualization with Texture

Volume rendering is a useful technique for visualizing three dimensional arrays of sampled data. Examples of sampled 3D data can range from computational fluid dynamics, medical data from CAT or MRI scanners, seismic data, or any volumetric information where geometric surfaces are difficult to generate or unavailable. Volume visualization provides a way to see through the data, revealing complex 3D relationships.

There are a number of approaches for visualization of volume data. Many of them use data analysis techniques to find the contour surfaces inside the volume of interest, then render the resulting geometry with transparency.

The 3D texture approach is a direct data visualization technique, using 2D or 3D textured data slices, combined using a blending operator [11]. The approach described here is equivalent to ray casting [22] and produces the same results. Unlike ray casting, where each image pixel is built up ray by ray, this approach takes advantage of spatial coherence. The 3D texture is used as a voxel cache, processing all rays simultaneously, one 2D layer at a time. Since an entire 2D slice of the voxels are "cast" at one time, the resulting algorithm is much more efficient than ray casting.

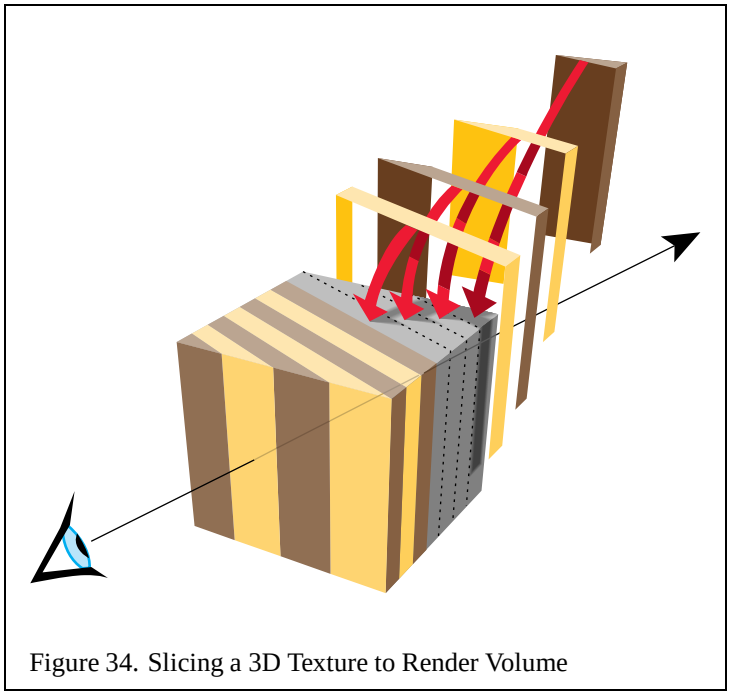


Figure 34. Slicing a 3D Texture to Render Volume

This section is divided into two approaches, one using 2D textures, the other using a 3D texture. Although the 3D texture approach is simpler and yields superior results overall, 3D textures are currently still an EXT extension in OpenGL and are not universally available like 2D textures. 3D texturing is currently slated to go into the core of OpenGL 1.2, so both methods [11] are described here.

13.1 Overview of the Technique

The technique for visualizing volume data is composed of two parts. First the texture data is sampled with planes parallel to the viewport and stacked along the direction of view. These planes are rendered as polygons, clipped to the limits of the texture volume. These clipped polygons are textured with the volume data, and the resulting images are blended together, from back to front, towards the viewing position. As each polygon is rendered, its pixel values are blended into the frame buffer to provide the appropriate transparency effect.

If the OpenGL implementation doesn't support 3D textures, a more limited version of the technique can be used, where 3 sets of 2D textures are created, one set for each major plane of the volume data. The process then proceeds as with the

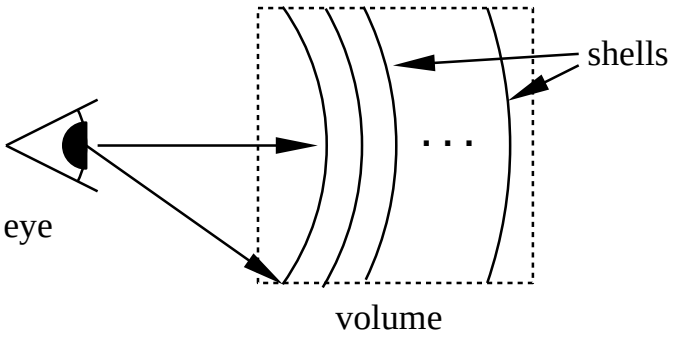


Figure 35. Slicing a 3D Texture with Spheres

3D case, except that the slices are constrained to be parallel to one of the three 2D texture sets.

Close-up views of the volume cause sampling errors to occur at texels that are far from the line of sight into the data. To correct this problem, use a series of concentric tessellated spheres centered around the eyepoint, rather than a single flat polygon, to generate each textured "slice" of the data. As with flat slices, the spherical shells should be clipped to the data volume, and each textured shell blended from back to front.

13.2 3D Texture Volume Rendering

Using 3D textures for volume rendering is the most desirable method. The slices can be oriented perpendicular to the viewer's line of sight, and close-up views can be rendered with spherical "shell slices" to avoid aliasing in the parts of the image that where sampled far from the direction of view.

Here are the steps for rendering a volume using 2D textures:

1. Load the volume data into a 3D texture. This is done once for a particular data volume.

2. Choose the number of slices, based on the criteria in Section 13.5. Usually this matches the texel dimensions of the volume data cube.

3. Find the desired viewpoint and view direction

4. Compute a series of polygons that cut through the data perpendicular to the direction of view. Use texture coordinate generation to texture the slice properly with respect to the 3D texture data.

5. Use the texture transform matrix to set the desired orientation the textured images on the slices.

6. Render each slice as a textured polygon, from back to front. A blend operation is performed at each slice; the type of blend depends on the desired effect. See the blend equation descriptions in Section 13.4 for details.

7. As the viewpoint and direction of view changes, recompute the data slice positions and update the texture transformation matrix as necessary.

13.3 2D Texture Volume Rendering

Volume rendering with 2D textures is more complex and does not provide as good results as 3D textures, but can be used on any OpenGL implementation.

The problem with 2D textures is that the data slice polygons can't always be perpendicular to the view direction. Three sets of 2D texture maps are created, each set perpendicular to one of the major axes of the data volume. These texture sets are created from adjacent 2D slices of the original 3D volume data along a major axis. The data slice polygons must be aligned with whichever set of 2D texture maps is most parallel to it. In the worst case, the data slices are canted 45 degrees from the view direction.

The more edge-on the slices are to the eye, the worse the data sampling is. In the extreme case of an edge-on slice, the textured values on the slices aren't blended at all. At each edge pixel, only one sample is visible, from the line of texel values crossing the polygon slice. All the other values are obscured.

For the same reason, sampling the texel data as spherical shells to avoid aliasing when doing close-ups of the volume data, isn't practical with 2D textures.

Here are the steps for rendering a volume using 2D textures:

1. Generate the three sets of 2D textures from the volume data. Each set of 2D textures is oriented perpendicular to one of volume's major axes. This processing is done once for a particular data volume.

2. Choose the number of slices, based on the criteria in Section 13.5. Usually this matches the texel dimensions of the volume data cube.

3. Find the desired viewpoint and view direction

4. Find the set of 2D textures most perpendicular to the direction of view. Generate data slice polygons parallel to the 2D texture set chosen. Use texture coordinate generation to texture each slice properly with respect to its corresponding 2D texture in the texture set.

5. Use the texture transform matrix to set the desired orientation the textured images on the slices.

6. Render each slice as a textured polygon, from back to front. A blend operation is performed at each slice; the type of blend depends on the desired effect. See the blend equation descriptions in Section 13.4 for details.

7. As the viewpoint and direction of view changes, recompute the data slice positions and update the texture transformation matrix as necessary. Always orient the data slices to the 2D texture set that is most closely aligned with it.

13.4 Blending Operators

There a number of common blending functions used in volume visualization. They are described below.

13.4.1 Over

The *over* operator [38] is the most common way to blend for volume visualization. Volumes blended with the over operator approximate the flow of light through a colored, translucent material. The translucency of each point in the material is determined by the value of the texel's alpha channel. Texels with higher alpha values tend to obscure texels behind them, and stand out through the obscuring texels in front of them.

The over operator can be implemented in OpenGL by setting the blend function to perform the over operation:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
```

13.4.2 Attenuate

The *attenuate* operator simulates an X-ray of the material. With attenuate, the texel's alpha appears to attenuate light shining through the material along the view direction towards the viewer. The texel alpha channel models material density. The final brightness at each pixel is attenuated by the total texel density along the direction of view.

Attenuation can be implemented with OpenGL by scaling each element by the number of slices, then summing the results. This can be done by combination of the appropriate blend function and blend color:

```
glBlendFunc(GL_CONSTANT_ALPHA_EXT, GL_ONE)
glBlendColorEXT(1.f, 1.f, 1.f, 1.f/number_of_slices)
```

13.4.3 MIP

In this context *MIP* stands for Maximum Intensity Projection. It is used in medical imaging to visualize blood flow. MIP finds the brightest texel alpha from all the texture slices at each pixel location. MIP is a contrast enhancing operator; structures with higher alpha values tend to stand out against the surrounding data.

MIP can be implemented with OpenGL using the blend function and the blend minmax extension:

```
glBlendFunc(GL_ONE, GL_ONE)
glBlendEquationEXT(GL_MAX_EXT)
```

13.4.4 Under

Volume slices rendered front to back with the *under* operator give the same result as the over operator blending slices from back to front. Unfortunately, OpenGL doesn't have an exact equivalent for the under operator, although using `glBlendFunc(GL_ONE_MINUS_DST, GL_DST)` is a good approximation. Use the over operator and back to front rendering for best results. See section 6.1 for more details.

13.5 Sampling Frequency

There are a number of factors to consider when choosing the number of slices (data polygons) to use when rendering your volume:

Performance It's often convenient to have separate "interactive" and "detail" modes for viewing volumes. The interactive mode can render the volume with a smaller number of slices, improving the interactivity at the expense of image quality. Detail mode - rendering with more slices - can be invoked when the volume being manipulated slows or stops.

Cubical Voxels The data slice spacing should be chosen so that the texture sampling rate from slice to slice is equal to the texture sampling rate within each

slice. Uniform sampling rate treats 3D texture texels as cubical voxels, which minimizes resampling artifacts.

For a cubical data volume, the number of slices through the volume should roughly match the resolution in texels of the slices. When the viewing direction is not along a major axis, the number of sample texels changes from plane to plane. Choosing the number of texels along each side is usually a good approximation.

Non-linear blending The over operator is not linear, so adding more slices doesn't just make the image more detailed. It also increases the overall attenuation, making it harder to see density details at the "back" of the volume. Strictly speaking, if you change the number of slices used to render the volume, the alpha values of the data should be rescaled. There is only one correct sample spacing for a given data set's alpha values. Generally, it doesn't buy you anything to have more slices than you have voxels in your 3D data.

Perspective When viewing a volume in perspective, the density of slices should increase with distance from the viewer. The data in the back of the volume should appear denser as a result of perspective distortion. If the volume isn't being viewed in perspective, then uniformly spaced data slices are usually the best approach.

Flat vs. Spherical Slices If you are using spherical slices to get good close-ups of the data, then the slice spacing should be handled in the same way as for flat slices. The spheres making up the slices should be tessellated finely enough to avoid concentric shells from touching each other.

2D vs. 3D Textures 3D textures can sample the data in the S, T, or R directions freely. 2D textures are constrained to S and T. 2D texture slices correspond exactly to texel slices of the volume data. To create a slice at an arbitrary point would require resampling the volume data.

Theoretically, the minimum data slice spacing is computed by finding the longest ray cast through the volume in the view direction, transforming the texel values found along that ray using the transfer function (if there is one), then finding the highest frequency component of the transformed texels, and using double that number for the minimum number of data slices for that view direction.

This can lead to a large number of slices. For a data cube 512 texels on a side, the worst case would be at least $1024\sqrt{3}$ slices, or about 1774 slices. In practice, however, the volume data tends to be bandwidth limited; and in many cases choosing the number of data slices to be equal to the volumes dimensions, measured in

texels, works well. In this example, you may get satisfactory results with 512 slices, rather than 1774. If the data is very blurry, or image quality is not paramount (for example, in "interactive mode"), this value could be reduced by a factor of two or four.

13.6 Shrinking the Volume Image

For best visual quality, render the volume image so that the size of a texel is about the size of a pixel. Besides making it easier to see density details in the image, larger images avoid the problems associated with under-sampling a minified volume.

Reducing the volume size will cause the texel data to be sampled to a smaller area. Since the over operator is non-linear, the shrunken data will interact with it to yield an image that is different, not just smaller. The minified image will have density artifacts that are not in the original volume data.

If a smaller image is desired, first render the image full size in the desired orientation, then shrink the resulting 2D image.

13.7 Virtualizing Texture Memory

Volume data doesn't have to be limited to the maximum size of 3D texture memory. The visualization technique can be virtualized by dividing the data volume into a set of smaller "bricks". Each brick is loaded into texture memory, then data slices are textured and blended from the brick as usual. The processing of bricks themselves is ordered from back to front relative to the viewer. The process is repeated with each brick in the volume until the entire volume has been processed.

To avoid sampling errors at the edges, data slice texture coordinates should be adjusted so they don't use the surface texels of any brick. The bricks themselves are oriented so that they overlap by one volume texel with their immediate neighbors. This allows the results of rendering each brick to combine seamlessly.

13.8 Mixing Volumetric and Geometric Objects

In many applications it is useful to display both geometric primitives and volumetric data sets in the same scene. For example, medical data can be rendered volumetrically, with a polygonal prosthesis placed inside it. The embedded geometry may be opaque or transparent.

Opaque geometric objects are rendered along with the volumetric data slice polygons using depth buffering for both. With depth buffering on, the pixels of planes behind the object aren't rendered, while the planes in front of the object blend

it in. The blending of the planes in front of the object gradually obscure it, making it appear embedded in the volume data.

If the object itself should be transparent, it must be rendered a slice at a time. The object is chopped into slabs using user defined clipping planes.The slab thickness corresponds to the spacing between volume data slices. Each slab of object corresponds to one of the data slices. Each slice of the object is rendered and blended with its corresponding data slice polygon, as the polygons are rendered back to front.

13.9 Transfer Functions

Different alpha values in volumetric data often correspond to different materials in the volume being rendered. To help analyze the volume data, a non-linear transfer function can be applied to the texels, highlighting particular classes of volume data. This transformation function can be applied through one of OpenGL's lookup tables. The SGI_texture_color_table extension applies a lookup table to texels values during texturing, after the texel value is filtered.

Since filtering adjusts the texel component values, a more accurate method is to apply the lookup table to the texel values before the textures are filtered. If the EXT_color_table table extension is available, then a colortable in the pixel path can be used to process the texel values while the texture is loaded. If lookup tables aren't available, the processing can be done to the volume data by the application, before loading the texture.

13.10 Volume Cutting Planes

Additional surfaces can be created on the volume with user defined clipping planes. A clipping plane can be used to cut through the volume, exposing a new surface. This technique can help expose the volume's internal structure. The rendering technique is the same, with the addition of one or more clipping planes defined while rendering and blending the data slice polygons.

13.11 Shading the Volume

In addition to visualizing the voxel data, the data can be lit and shaded. Since there are no explicit surfaces in the data, lighting is computed per volume texel.

The direct approach to shading is to do it on the host. The volumetric data can be processed to find the gradient at each voxel. Then the dot product between the gradient vector, now used as a normal, and the light is computed, and the results saved as 3D data. The volumetric data now contains the intensity at each point in the

data, instead of data density. Specular intensity can be computed the same way, and combined so that each texel contains the total light intensity at every sample point in the volume. This processed data can then be visualized in the manner described previously.

The problem with this technique is that a change of light source (or viewer position, if specular lighting is desired) requires that the data volume be reprocessed. A more flexible approach is to save the components of the gradient vectors as color components in the 3D texture. Then the lighting can be done while the data is being visualized. One way to do this is to transform the texel data using the color matrix extension. The light direction can be processed to form a matrix that when multiplied by the texture color components (now containing the components of the normal at that point), the will produce the dot product of the two. The color matrix is part of the pixel path, so this processing can be done when the texture is being loaded. Now the 3D texture contains lighting intensities as before, but the dot product calculations are done in the pixel pipeline, not in the host.

The data's gradient vectors could also be computed interactively, as an extension of the texture bump-mapping technique described in Section 8.3. Each data slice polygon is treated as a surface polygon to be bump-mapped. Since the texture data must be shifted and subtracted, then blended with the shaded polygon to generate the lit slice before blending, the process of generating lit slices must be processed separately from the blending of slices to create the volume image.

13.12 Warped Volumes

The data volume can be warped by non-linear shifting the texture coordinates of the data slices. For more warping control, tessellate the the vertices to provide more vertex locations to perturb the texture coordinate values. Among other things, very high quality atmospheric effects, such as smoke, can be produced with this technique.

14 Using the Stencil Buffer

The stencil buffer is like the depth and color buffers, but is a value per pixel that has an application-specific use. The stencil buffer isn't directly visible like the color buffer, but the bits in the stencil planes form an unsigned integer that affects and is updated by drawing commands, through the stencil function and the stencil operations. The stencil function controls whether a fragment is discarded or not by the stencil test, and the stencil operation determines how the stencil planes are updated as a result of that test. [32].

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

Table 4: Stencil Buffer Comparisons

Stencil buffer actions are part of OpenGL's fragment operations. Stencil testing occurs immediately after the alpha test, and immediately before the depth test. If GL_STENCIL_TEST is enabled, and stencil planes are available, the application can control what happens under three different scenarios:

1. The stencil test fails
2. The stencil test passes, but the depth test fails
3. Both the stencil and the depth test pass.

Whether a stencil operation for a given fragment passes or fails has nothing to do with the color or depth value of the fragment. The stencil operation is a comparison between the value in the stencil buffer for the fragment's destination pixel and the stencil reference value. A mask is bitwise AND-ed with the value in the stencil planes and with the reference value before the the comparison is applied. The reference value, the comparison function, and the comparison mask are set by glStencilFunc. The comparison functions available are listed in Table 4.

Stencil function and *stencil test* are often used interchangeably in these notes, but the "stencil test" specifically means the application of the stencil function in conjunction with the stencil mask.

If the stencil test fails, the fragment is discarded (the color and depth values for that pixel remain unchanged) and 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 the depth test passes (or if depth testing is disabled or if the visual does not have a depth buffer), the fragment continues on through 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.

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

Table 5: Stencil Buffer Operations

Thus, the stencil test controls which fragments continue towards the framebuffer, and the stencil operation controls how the stencil buffer is updated by the results of both the stencil test and the depth test.

The stencil operations available are described in Table 5.

The `glStencilOp` call sets the stencil operations for all three stencil test results: stencil fail, stencil pass/depth buffer fail, and stencil pass/depth buffer pass.

Writes to the stencil buffer can be disabled and enabled per bit by `glStencilMask`. This allows an application to apply stencil tests without the results affecting the stencil values, or to partition the stencil buffer into several smaller logical stencil buffers. Keep in mind, however, that the `GL_INCR` and `GL_DECR` operations operate on each stencil value as a whole, and may not operate as expected when the stencil mask is not all ones. Stencil writes can also be disabled by calling `glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP)`.

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

Different machines support different numbers of stencil bits per pixel. Use `glGetIntegerv(GL_STENCIL_BITS, ...)` to see how many bits the visual supports. If multiple stencil bits are available, the mask argument to `glStencilFunc` can be used to divide up the stencil buffer into a number of different sections. This allows the application to store separate stencil values per pixel within the same stencil buffer.

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

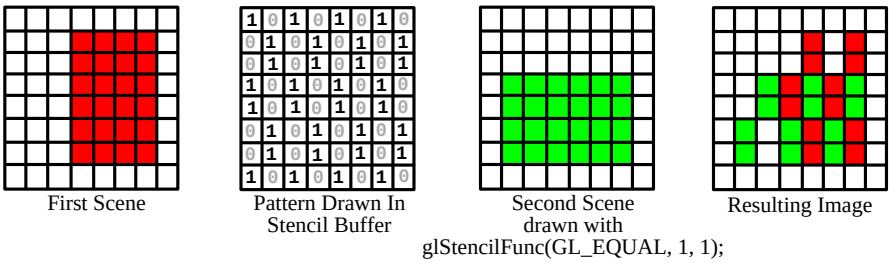


Figure 36. Using stencil to dissolve between images

14.1 Dissolves with Stencil

Stencil buffers can be used to mask selected pixels on the screen. This allows for pixel by pixel compositing of images. You can draw geometry or arrays of stencil values to control, per pixel, what is drawn into the color buffer. One way to use this capability is 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 alpha planes of the color buffer and the alpha function can also be used to implement this kind of dissolve, but using the stencil buffer frees up the alpha planes for motion blur, transparency, smoothing, and other effects.

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 images’ reference values, and fail with the other. An example of a dissolve partway between two images is shown in Figure 36.

At the start of the dissolve (the first frame of the sequence), the stencil buffer is all cleared to 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 only when compared against the second image’s reference value. 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(GL_STENCIL_BUFFER_BIT)`
2. Disable writing to the color buffer, using `glColorMask(GL_FALSE,GL_FALSE,GL_FALSE,GL_FALSE)`
3. If the values in the depth buffer should not change, use `glDepthMask(GL_FALSE)`

For this example, we'll have the stencil test always fail, and set the stencil operation to write the reference value to the stencil buffer. Your application will also need to turn on stenciling before you begin drawing the dissolve pattern.

1. Turn on stenciling; `glEnable(GL_STENCIL_TEST)`
2. Set stencil function to always fail; `glStencilFunc(GL_NEVER, 1, 1)`
3. Set stencil op to write 1 on stencil test failure; `glStencilOp(GL_REPLACE, GL_KEEP, GL_KEEP)`
4. Write the dissolve pattern to the stencil buffer by drawing geometry or using `glDrawPixels`.
5. Disable writing to the stencil buffer with `glStencilMask(GL_FALSE)`.
6. Set stencil function to pass on 0; `glStencilFunc(GL_EQUAL, 0, 1)`.
7. Enable color buffer for writing with `glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE)`.
8. If you're depth testing, turn depth buffer writes back on with `glDepthMask`.
9. Draw the first image. It will only be written where the stencil buffer values are 0.
10. Change the stencil test so only values that are 1 pass; `glStencilFunc(GL_EQUAL, 1, 1)`.
11. Draw the second image. Only pixels with stencil value of 1 will change.
12. 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 each new frame's dissolve pattern is a superset of the previous frame's pattern, image 1 doesn't have to be re-rendered. This is because once a pixel of image 1 is replaced with image 2, image 1 will never be redrawn there. Designing the dissolve pattern with this restriction can improve the performance of this technique.

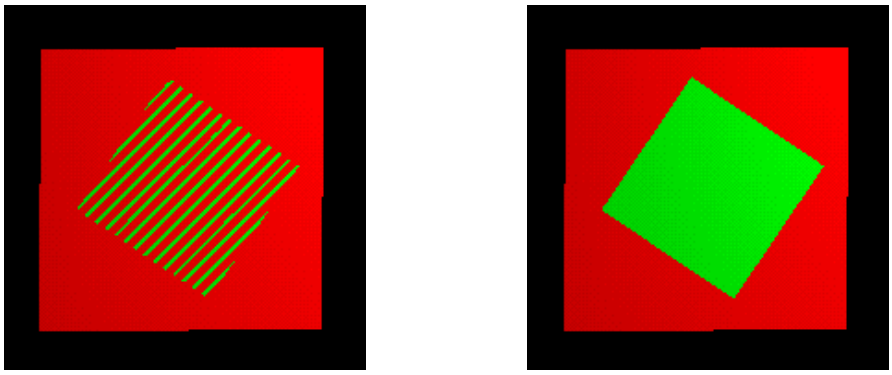


Figure 37. Using stencil to render coplanar polygons

14.2 Decaling with Stencil

In the dissolve example, the stencil buffer controls where pixels were drawn from an entire scene. Using stencil to control pixels drawn from a particular primitive 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 directly on top of another (runway markings, for example), the depth buffer can't be relied upon to produce a clean separation between the two. This is due to the quantization 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, so some pixels of the back polygon may show through the front polygon. In an application with a high frame rate, this results in a shimmering mixture of pixels from both polygons (commonly called "Z fighting" or "flimmering"). An example is shown in in Figure 37.

To solve this problem, the closer polygons are drawn with the depth test disabled, on the same pixels covered by the farthest polygons. It appears that the closer

polygons are “decaled” on the farther polygons.

Decaled polygons can be drawn 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; the stencil values are initialized 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, and zero where the base polygon generated fragments that failed the depth test. 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 there are other primitives partially obscuring the base polygon and decal polygons.

There are a few limitations to this technique. First, it assumes that the decal polygon 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, if the screen extents of the base polygons you’re decaling overlap, you’ll have to perform the decal process for one base polygon and its decals before you move on to another base and decals. This is an important consideration if your application collects and then sorts geometry based on its graphics state, where the rendering order of geometry may be changed by the sort.

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 the following 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.

Multiple textures can be drawn onto a polygon with a similar technique. Instead of writing decal polygons, the same polygon is drawn with each subsequent texture and an alpha value to blend the old pixel color and the new pixel color together.

14.3 Finding Depth Complexity with the Stencil Buffer

Finding depth complexity, or how many fragments were generated for each pixel in a depth buffered scene, is important for analyzing graphics performance. It indicates how well polygons are distributed across the frame buffer and how many fragments were generated and discarded, clues for application tuning.

One way to show depth complexity is to use the color values of the pixels in the scene to indicate the number of times a pixel was written. It is relatively easy to draw an image representing depth complexity with the stencil buffer. The basic approach is simple. Increment a pixel's stencil value every time the 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.

This technique generates a count of the number of fragments generated for each pixel, whether the depth test failed or not. By changing the stencil operations, a similar technique could be used to count the number of fragments discarded after failing the depth test or to count the number of times a pixel was covered by fragments passing the depth test.

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 of stencil values to colors by turning on the color mapping with `glPixelTransferi(GL_MAP_COLOR, GL_TRUE)` and setting the appropriate pixel transfer maps with `glPixelMap`. You can map the stencil values to either RGBA or color index values, depending on the type of color buffer to which you're writing.

14.4 Compositing Images with Depth

Compositing separate images together is a useful technique for increasing the complexity of a scene [12]. An image can be saved to memory, then drawn to the screen using `glDrawPixels`. Both the color and depth buffer contents can be copied into the frame buffer. This is sufficient for 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 for 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. `glStencilOp` is called 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 will have been merged, using the depth test to control which pixels from the saved image would 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.

15 Line Rendering Techniques

15.1 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 [27] [4].

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 may show depth buffer aliasing artifacts), use polygon offset or stencil decaling techniques to draw the polygon edges. The following technique works well, although its not completely general. Use the stencil buffer to mask where all the lines, both hidden and visible, are. Then use the stencil function to prevent the polygon rendering from updating 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 from effecting the hidden lines.

15.2 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 can be drawn using the depth buffer. The technique has two passes. First disable writing to the color buffer; the first pass only updates 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_LEQUAL
8. Render lines

This method will not work where multiple lines with the same depth meet. Instead of connecting, all of the lines will be "blocked" by the last wide line drawn. There can also be depth buffer aliasing problems when the wide line z values are changed by another wide line crossing it. This effect becomes more pronounced if the narrow lines are widened to improve image clarity.

To avoid this problem, use polygon offset to move the narrower visible lines in front of the obscuring lines. The minimum offset should be used to avoid lines from one surface of the object "popping through" the lines of a another surface separated by only a small depth value.

If the vertices of the objects faces are oriented to allow face culling, Then face culling can be used to sort the object surfaces and allow a more robust technique: The lines of the objects back faces are drawn, then obscuring wide lines of the front face are drawn, then finally the narrow lines of the front face are drawn. No special depth buffer techniques are needed.

1. Cull the front faces of the object
2. Draw the object as lines
3. Cull the back faces of the object
4. Draw the object as wide lines in the background color
5. Draw the object as lines

Since the depth buffer isn't needed, there are no depth aliasing problems. The backface culling technique is fast and works well, but is not general. It won't work for multiple obscuring or intersecting objects.

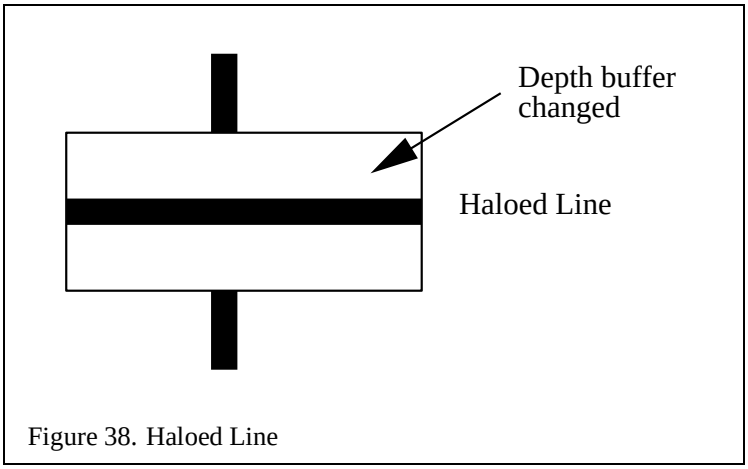


Figure 38. Haloed Line

15.3 Silhouette Edges

Sometimes it can be useful for highlighting purposes to draw a silhouette edge around a complex object. A silhouette edge defines the outer boundaries of the object with respect to the viewer.

The stencil buffer can be used to render a silhouette edge around an object. With this technique, you can render the object, then draw a silhouette around it, or just draw the silhouette itself [40].

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.

16 Tuning Your OpenGL Application

Tuning your software makes it use hardware capabilities more effectively. Writing high-performance code is usually more complex than just following a set of rules. Most often, it involves making trade-offs between special functionality, quality, and performance for a particular application.

16.1 What Is Pipeline Tuning?

Traditional software tuning focuses on finding and tuning hot spots, the 10% of the code in which a program spends 90% of its time. Pipeline tuning uses a different approach: it looks for bottlenecks, overloaded stages that are holding up other processes.

At any time, one stage of the pipeline is the bottleneck. Reducing the time spent in the bottleneck is the best way to improve performance. Conversely, doing work that further narrows the bottleneck, or that creates a new bottleneck somewhere else, will further degrade performance. If different parts of the hardware are responsible

for different parts of the pipeline, the workload can be increased at other parts of the pipeline without degrading performance, as long as that part does not become a new bottleneck. In this way, an application can sometimes be altered to draw a higher-quality image with no performance degradation.

Different programs stress different parts of the pipeline, so it's important to understand which elements in the graphics pipeline are the bottlenecks for your program.

Note that in a software implementation, all the work is done on the host CPU. As a result, it doesn't make sense to increase the work in the geometry pipeline if rasterization is the bottleneck: you'd be increasing the work for the CPU and decreasing performance.

16.1.1 Three-Stage Model of the Graphics Pipeline

The graphics pipeline consists of three conceptual stages. Depending on the implementation, all parts may be done by the CPU or parts of the pipeline may be done by an accelerator card. The conceptual model is useful in either case: it helps you to know where your application spends its time. The stages are:

- The application. The application program running on the CPU, feeding commands to the graphics subsystem (always on the CPU).
- The geometry subsystem. The per-polygon operations, such as coordinate transformations, lighting, texture coordinate generation, and clipping (may be hardware-accelerated).
- The raster subsystem. The per-pixel operations, such as the simple operation of writing color values into the framebuffer, or more complex operations like depth buffering, alpha blending, and texture mapping (may be hardware accelerated).

The amount of work required from the different pipeline stages varies depending on what the application does. For example, consider a program that draws a small number of large polygons. Because there are only a few polygons, the pipeline stage that does geometry operations is lightly loaded. Because those few polygons cover many pixels on the screen, the pipeline stage that does rasterization is heavily loaded.

To speed up this program, you must speed up the rasterization stage, either by drawing fewer pixels, or by drawing pixels in a way that takes less time by turning off modes like texturing, blending, or depth-buffering. In addition, because spare

capacity is available in the per-polygon stage, you may be able to increase the workload at that stage without degrading performance. For example, try to use a more complex lighting model, or define geometries such that they remain the same size but look more detailed because they are composed of a larger number of polygons.

16.1.2 Finding Bottlenecks in Your Application

The basic strategy for isolating bottlenecks is to measure the time it takes to execute a program (or part of a program) and then change the code in ways that don't alter its performance (except by adding or subtracting work at a single point in the graphics pipeline). If changing the amount of work at a given stage of the pipeline does not alter performance appreciably, that stage is not the bottleneck. If there is a noticeable difference in performance, you've found a bottleneck.

Application bottlenecks. To see if your application is the bottleneck, remove as much graphics work as possible, while preserving the behavior of the application in terms of the number of instructions executed and the way memory is accessed. Often, changing just a few OpenGL calls is a sufficient test. For example, replacing the vertex and normal calls `glVertex3fv` and `glNormal3fv` with color subroutine calls (`glColor3fv`) preserves the CPU behavior while eliminating all drawing and lighting work in the graphics pipeline. If making these changes does not significantly improve performance, then your application is the bottleneck.

Geometry bottlenecks. Programs that create bottlenecks in the geometry (per-polygon) stage are termed *transform limited*. To test for bottlenecks in geometry operations, change the program so that the application code runs at the same speed and the same number of pixels are filled, but the geometry work is reduced. For example, if you are using lighting, call `glDisable` with a `GL_LIGHTING` argument to temporarily turn off lighting. If performance improves, your application has a per-polygon bottleneck. For more information, see "Tuning the Geometry Subsystem".

Rasterization bottlenecks. Programs that cause bottlenecks at the rasterization (per-pixel) stage in the pipeline are *fillrate limited*. To test for bottlenecks in rasterization operations, shrink objects or make the window smaller to reduce the number of active pixels. This technique won't work if your program alters its behavior based on the sizes of objects or the size of the window. You can also reduce the work done per pixel by turning off per-pixel operations such as depth-buffering, texturing, or alpha blending. If any of these experiments speed up the program, it has a fill-rate bottleneck. For more information, see "Tuning the Raster Subsystem".

Performance Parameter	Pipeline Stage
Amount of data per polygon	All stages
Time of application overhead	Application
Transform rate and mode setting for polygon	Geometry subsystem
Total number of polygons in a frame	Geometry and raster subsystem
Number of pixels filled	Raster subsystem
Fill rate for the given mode settings	Raster subsystem
Time of screen and/or depth buffer clear	Raster subsystem

Table 6: Factors Influencing Performance

Many programs draw a variety of things, each of which stresses different parts of the system. Decompose such a program into pieces and time each piece. You can then focus on tuning the slowest pieces.

16.1.3 Factors Influencing Performance

Table 6 provides an overview of factors that may limit rendering performance and the part of the pipeline they belong to.

16.2 Optimizing Your Application Code

16.2.1 Optimize Cache and Memory Usage

On most systems, memory is structured as a hierarchy that contains a small amount of faster, more expensive memory at the top and a large amount of slower memory at the base. The hierarchy is organized from registers in the CPU at the top down to the disks at the bottom. As memory locations are referenced, they are automatically copied into higher levels of the hierarchy, so data that is referenced most often migrates to the fastest memory locations.

The goal of machine designers and programmers is to maximize the chance of finding data as high up in the memory hierarchy as possible. To achieve this goal, algorithms for maintaining the hierarchy, embodied in the hardware and the operating system, assume that programs have locality of reference in both time and space; that is, programs keep frequently accessed locations close together. Performance increases if you respect the degree of locality required by each level in the memory hierarchy.

Minimizing Cache Misses. Most CPU's have first-level instruction and data caches on chip and many have second-level cache(s) that are bigger but somewhat slower. Memory accesses are much faster if the data is already loaded into the first-level cache. When your program accesses data that isn't in one of the caches, it gets a *cache miss*. This causes a block of consecutively addressed words, including the data that you just accessed, to be loaded into the cache. Since cache misses are costly, you should try to minimize them, using these tips:

- Keep frequently accessed data together. Store and access frequently used data in flat, sequential data structures and avoid pointer indirection. This way, the most frequently accessed data remains in the first-level cache as much as possible.
- Access data sequentially. Each cache miss brings in a block of consecutively addressed words of needed data. If you are accessing data sequentially then each cache miss will bring in n words (where n is system dependent); if you are accessing only every nth word, then you will be constantly reading in unneeded data, degrading performance.
- Avoid simultaneously traversing several large buffers of data, such as an array of vertex coordinates and an array of colors within a loop since there can be cache conflicts between the buffers. Instead, pack the contents into one buffer whenever possible. If you are using vertex arrays, try to use interleaved arrays. (For more information on vertex arrays see "Rendering Geometry Efficiently".) However, if packing your data forces a big increase in the size of the data, it may not be the right optimization for your program.

16.2.2 Store Data in a Format That is Efficient for Rendering

Putting some extra effort into generating a simpler database makes a significant difference when traversing that data for display. A common tendency is to leave the data in a format that is good for loading or generating the object, but non-optimal for actually displaying it. For peak performance, do as much of the work as possible before rendering. Preprocessing of data is typically done at initialization time or when changing from a modeling to a fast-rendering mode.

See "Rendering Geometry Efficiently" and "Rendering Images Efficiently" for tips on how to store your geometric data and image data to make it more efficient for rendering.

Minimizing State Changes. Your program will almost always benefit if you reduce the number of state changes. A good way to do this is to sort your scene data

according to what state is set and render primitives with the same state settings together. Primitives should be sorted by the most expensive state settings first. Typically it is expensive to change texture binding, material parameters, fog parameters, texture filter modes, and the lighting model. However, some experimentation will be required to determine which state settings are most expensive on the system you are running on. For example, on systems that accelerate rasterization, it may not be that expensive to change rasterization controls such as the depth test function and whether or not depth testing is enabled. But if you are running on a system with software rasterization, this may cause the graphics pipeline to be revalidated.

It is also important to avoid redundant state changes. If your data is stored in a hierarchical database, make decisions about which geometry to display and which modes to use at the highest possible level. Decisions that are made too far down the tree can be redundant.

16.2.3 Per-Platform Tuning

Many of the performance tuning techniques discussed here (e.g., minimizing the number of state changes and disabling features that aren't required) are a good idea no matter what system you are running on. Other tuning techniques need to be programmed for a particular system. For example, before you sort your database based on state changes, you need to determine which state changes are the most expensive for each system you are interested in running on.

In addition, you may want to modify the behavior of your program depending on which modes are fast. This is especially important for programs that must run at a particular frame rate. Features may need to be disabled in order to maintain the frame rate. For example, if a particular texture mapping environment is slow on one of your target systems, you may need to disable texture mapping or change the texture environment whenever your program is running on that platform.

Before you can tune your program for each of the target platforms, you will need to do some performance characterization. This isn't always straightforward. Often a particular device is able to accelerate certain features, but not all at the same time. Thus it is important to test the performance for combinations of features that you will be using. For example, a graphics adapter may accelerate texture mapping but only for certain texture parameters and texture environment settings. Even if all texture modes are accelerated, experimentation will be required to see how many textures you can use at once without causing the adapter to page textures in and out of the local memory.

An even more complicated situation arises if the graphics adapter has a shared pool of memory that is allocated to several tasks. For example, the adapter may not have a frame buffer deep enough to contain a depth buffer and a stencil buffer. In

this case, the adapter would be able to accelerate both depth buffering and stenciling but not at the same time. Or perhaps, depth buffering and stenciling can both be accelerated but only for certain stencil buffer depths.

Typically, per-platform testing is done at initialization time. You should do some trial runs through your data with different combinations of state settings and calculate the time it takes to render in each case. You may want to save the results in a file so your program doesn't have to do this each time it starts up. You can find an example of how to measure the performance of particular OpenGL operations and save the results using the isfast program on the website.

16.3 Tuning the Geometry Subsystem

16.3.1 Use Expensive Modes Efficiently

OpenGL offers many features that create sophisticated effects with excellent performance. However, these features have some performance cost, compared to drawing the same scene without them. Use these features only where their effects, performance, and quality are justified.

- Turn off features when they are not required. Once a feature has been turned on, it can slow the transform rate even when it has no visible effect.

  For example, the use of fog can slow the transform rate of polygons even when the polygons are too close to show fog, and even when the fog density is set to zero. For these conditions, turn off fog explicitly with `glDisable(GL_FOG)`.

- Minimize mode changes. Be especially careful about expensive mode changes such as changing `glDepthRange` parameters and changing fog parameters when fog is enabled.

- For optimum performance, use flat shading whenever possible. This reduces the number of lighting computations from one per-vertex to one per-primitive, and also reduces the amount of data that must be processed for each primitive. This is particularly important for high-performance line drawing.

16.3.2 Optimizing Transformations

OpenGL implementations are often able to optimize transform operations if the matrix type is known. Follow these guidelines to achieve optimal transform rates:

- Use `glLoadIdentity` to initialize a matrix, rather than loading your own copy of the identity matrix.

- Use specific matrix calls such as `glRotate`, `glTranslate`, and `glScale` rather than composing your own rotation, translation, or scale matrices and calling `glLoadMatrix` and/or `glMultMatrix`.

16.3.3 Optimizing Lighting Performance

OpenGL offers a large selection of lighting features. The penalties some features carry may vary depending on the hardware you're running on. Be prepared to experiment with the lighting configuration.

As a general rule, use the simplest possible lighting model: a single infinite light with an infinite viewer. For some local effects, try replacing local lights with infinite lights and a local viewer.

Use the following settings for peak performance lighting:

- Single infinite light.
- Nonlocal viewing. Set `GL_LIGHT_MODEL_LOCAL_VIEWER` to `GL_FALSE` in `glLightModel`. (the default)
- Single-sided lighting. Set `GL_LIGHT_MODEL_TWO_SIDE` to `GL_FALSE` in `glLightModel`. (the default)
- Disable `GL_COLOR_MATERIAL`.
- Disable `GL_NORMALIZE`. Since it is usually necessary to renormalize normals when the model-view matrix includes a scaling transformation, consider preprocessing the scene to eliminate scaling.

In addition, follow these guidelines to achieve peak lighting performance:

- Avoid using multiple lights.

  There may be a sharp drop in lighting performance when adding lights.

- Avoid using local lights.

  Local lights are noticeably more expensive than infinite lights.

- Don't change material parameters frequently.

  Changing material parameters can be expensive. If you need to change the material parameters many times per frame, consider rearranging the scene traversal to minimize material changes. Also consider using `glColorMaterial` if you need to change some material parameters often, rather than using `glMaterial` to change parameters explicitly.

The following code fragment illustrates how to change ambient and diffuse material parameters at every polygon or at every vertex:

```
glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);
glEnable(GL_COLOR_MATERIAL);
/* Draw triangles: */
glBegin(GL_TRIANGLES);
/* Set ambient and diffuse material parameters: */
glColor4f(red, green, blue, alpha);
glVertex3fv(...);glVertex3fv(...);glVertex3fv(...);
glColor4f(red, green, blue, alpha);
glVertex3fv(...);glVertex3fv(...);glVertex3fv(...);
...
glEnd();
```

- Avoid local viewer.

  Local viewing: Setting `GL_LIGHT_MODEL_LOCAL_VIEWER` to `GL_TRUE` with `glLightModel`, while using infinite lights only, reduces performance by a small amount. However, each additional local light noticeably degrades the transform rate.

- Disable two-sided lighting.

  Two-sided lighting illuminates both sides of a polygon. This is much faster than the alternative of drawing polygons twice. However, using two-sided lighting is significantly slower than one-sided lighting for a single rendering of an object.

- Disable `GL_NORMALIZE`.

  If possible, provide unit-length normals and don't call `glScale` to avoid the overhead of `GL_NORMALIZE`. On some OpenGL implementations it may be faster to simply rescale the normal, instead of renormalizing it, when the modelview matrix contains a uniform scale matrix. The `EXT_rescale_normal` extension may be supported by these implementations to improve the performance of this case. If so, you can enable `GL_RESCALE_NORMAL_EXT` and the normal will be rescaled making renormalization unnecessary.

- Avoid changing the `GL_SHININESS` material parameter if possible.

  Setting a new `GL_SHININESS` value requires significant computation each time.

- Avoid using lighting calls inside a `glBegin`/`glEnd` sequence.
- If possible, avoid calls to `glMaterial` during a `glBegin`/`glEnd` drawing sequence.

  Calling `glMaterial` between `glBegin`/`glEnd` has a serious performance impact. While making such calls to change colors by changing material properties is possible, the performance penalty makes it unadvisable. Use `glColorMaterial` instead.

16.3.4 Advanced Geometry-Limited Tuning Techniques

This section describes advanced techniques for tuning transform-limited drawing. Follow these guidelines to draw objects with complex surface characteristics:

- Use texture to replace complex geometry.

  Texture mapping can be used instead of extra polygons to add detail to a geometric object. This can greatly simplify geometry, resulting in a net speed increase and an improved picture, as long as it does not cause the program to become fill-limited.

- Use textured polygons as single-polygon billboards.

  Billboards are polygons that are fixed at a point and rotated about an axis, or about a point, so that the polygon always faces the viewer. Billboards can be used for distant objects to save geometry.

- Use `glAlphaFunc` in conjunction with one or more textures to give the effect of rather complex geometry on a single polygon.

  Consider drawing an image of a complex object by texturing it onto a single polygon. Set alpha values to zero in the texture outside the image of the object. (The edges of the object can be antialiased by using alpha values between zero and one.) Orient the polygon to face the viewer. To prevent pixels with zero alpha values in the textured polygon from being drawn, call `glAlphaFunc(GL_NOTEQUAL, 0.0)`.

  This effect is often used to create objects like trees that have complex edges or many holes through which the background should be visible (or both).

- Eliminate objects or polygons that will be out of sight or too small to see.
- Use fog to increase visual detail without drawing small background objects.

16.4 Tuning the Raster Subsystem

An explosion of both data and operations is required to rasterize a polygon as individual pixels. Typically, the operations include depth comparison, Gouraud shading, color blending, logical operations, texture mapping, and possibly antialiasing. The following techniques can improve performance for a fill-limited applications.

16.4.1 Using Backface/Frontface Removal

To reduce fill-limited drawing, use backface and frontface removal. For example, if you are drawing a sphere, half of its polygons are backfacing at any given time. Backface and frontface removal is done after transformation calculations but before per-fragment operations. This means that backface removal may make transform-limited polygons somewhat slower, but make fill-limited polygons significantly faster. You can turn on backface removal when you are drawing an object with many backfacing polygons, then turn it off again when drawing is completed.

16.4.2 Minimizing Per-Pixel Calculations

Another way to improve fill-limited drawing is to reduce the work required to render fragments.

Avoid Unnecessary Per-Fragment Operations. Turn off per-fragment operations for objects that do not require them, and structure the drawing process to minimize their use without causing excessive toggling of modes. For example, if you are using alpha blending to draw some partially transparent objects, make sure that you disable blending when drawing the opaque objects. Also, if you enable alpha test to render textures with holes through which the background can be seen, be sure to disable alpha testing when rendering textures or objects with no holes. It also helps to sort primitives so that primitives that require alpha blending or alpha test to be enabled, are drawn at the same time.

Use Simple Fill Algorithms for Large Polygons. If you are drawing very large polygons such as “backgrounds”, your performance will be improved if you use simple fill algorithms. For example, you should set `glShadeModel` to `GL_FLAT` if smooth shading isn’t required. Also, disable per-fragment operations such as depth buffering, if possible. If you need to texture the background polygons, consider using `GL_REPLACE` for the texture environment.

Use the Depth Buffer Efficiently. Any rendering operation can become fill-limited for large polygons. Clever structuring of drawing can eliminate or minimize per-pixel depth buffering operations. For example, if large backgrounds are drawn first, they do not need to be depth buffered. It is better to disable depth buffering for the backgrounds and then enable it for other objects where it is needed.

Games and flight simulators often use this technique. The sky and ground are drawn with depth buffering disabled, then the polygons lying flat on the ground (runway and grid) are drawn without suffering a performance penalty. Finally, depth buffering is enabled for drawing the mountains and airplanes.

There are many other special cases in which depth buffering might not be required. For example, terrain, ocean waves, and 3D function plots are often represented as height fields (X-Y grids with one height value at each lattice point). It's straightforward to draw height fields in back-to-front order by determining which edge of the field is furthest away from the viewer, then drawing strips of triangles or quadrilaterals parallel to that starting edge and working forward. The entire height field can be drawn without depth testing provided it doesn't intersect any piece of previously-drawn geometry. Depth values need not be written at all, unless subsequently-drawn depth buffered geometry might intersect the height field; in that case, depth values for the height field should be written, but the depth test can be avoided by calling `glDepthFunc(GL_ALWAYS)`.

16.4.3 Optimizing Texture Mapping

Follow these guidelines when rendering textured objects:

- Avoid frequent switching between texture maps. If you have many small textures, consider combining them into a single larger, tiled texture. Rather than switching to a new texture before drawing a textured polygon choose texture coordinates that select the appropriate small texture tile within the large texture.
- Use texture objects to encapsulate texture data. Place all the `glTexImage` calls (including mipmaps) required to completely specify a texture and the associated `glTexParameter` calls (which set texture properties) into a texture object and bind this texture object to the rendering context. This allows the implementation to compile the texture into a format that is optimal for rendering and, if the system accelerates texturing, to efficiently manage textures on the graphics adapter.
- If possible, use `glTexSubImageD` to replace all or part of an existing texture image rather than the more costly operations of deleting and creating an entire new image.

- Call `glAreTexturesResident` to make sure that all your textures are resident during rendering. (On systems where texturing is done on the host, `glAreTexturesResident` always returns `GL_TRUE`.) If necessary, reduce the size or internal format resolution of your textures until they all fit into memory. If such a reduction creates intolerably fuzzy textured objects, you may give some textures lower priority.

- Avoid expensive texture filter modes. On some systems, trilinear filtering is much more expensive than point sampling or bilinear filtering.

16.4.4 Clearing the Color and Depth Buffers Simultaneously

The most basic per-frame operations are clearing the color and depth buffers. On some systems, there are optimizations for common special cases of these operations.

Whenever you need to clear both the color and depth buffers, don't clear each buffer independently. Instead use `glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)`

Also, be sure to disable dithering before clearing.

16.5 Rendering Geometry Efficiently

16.5.1 Using Peak-Performance Primitives

This section describes how to draw geometry with optimal primitives. Consider these guidelines to optimize drawing:

- Use connected primitives (line strips, triangle strips, triangle fans, and quad strips).

  Connected primitives are desirable because they reduce the amount of data and the amount of per-polygon or per-line work done by the OpenGL. Be sure to put as many vertices as possible in a `glBegin`/`glEnd` sequence to amortize the cost of a `glBegin` and `glEnd`.

- Avoid using `glBegin(GL_POLYGON)`.

  When rendering independent triangles, use `glBegin(GL_TRIANGLES)` instead of `glBegin(GL_POLYGON)`. Also, when rendering independent quadrilaterals, use `glBegin(GL_QUADS)`.

- Batch primitives between `glBegin` and `glEnd`.

Use a single call to `glBegin(GL_TRIANGLES)` to draw multiple independent triangles rather than calling `glBegin(GL_TRIANGLES)` multiple times. Also, use a single call to `glBegin(GL_QUADS)` to draw multiple independent quadrilaterals, and a single call to `glBegin(GL_LINES)` to draw multiple independent line segments.

- Use "well-behaved" polygons–convex and planar, with only three or four vertices.

  Concave and self-intersecting polygons must be tessellated by GLU before they can be drawn, and are therefore prohibitively expensive. Nonplanar polygons and polygons with large numbers of vertices are more likely to exhibit shading artifacts.

  If your database has polygons that are not well-behaved, perform an initial one-time pass over the database to transform the troublemakers into well-behaved polygons and use the new database for rendering. You can store the results in OpenGL display lists. Using connected primitives results in additional gains.

- Minimize the data sent per vertex.

  Polygon rates can be affected directly by the number of normals or colors sent per polygon. Setting a color or normal per vertex, regardless of the `glShadeModel` used, may be slower than setting only a color per polygon, because of the time spent sending the extra data and resetting the current color. The number of normals and colors per polygon also directly affects the size of a display list containing the object.

- Group like primitives and minimize state changes to reduce pipeline revalidation.

16.5.2 Using Vertex Arrays

Vertex arrays are available in OpenGL 1.1. They offer the following benefits:

- The OpenGL implementation can take advantage of uniform data formats.
- The `glInterleavedArrays` call lets you specify packed vertex data easily. Packed vertex formats are typically faster for OpenGL to process.
- The `glDrawArrays` call reduces subroutine call overhead.
- The `glDrawElements` call reduces subroutine call overhead and also reduces per-vertex calculations because vertices are reused.

- Use the `EXT_compiled_vertex_array` extension if it is available. This extension allows you to lock down the portions of the arrays that you are using. This way the OpenGL implementation can DMA the arrays to the graphics adapter or reuse per-vertex calculations for vertices that are shared by adjacent primitives.

If you use `glBegin` and `glEnd` instead of `glDrawArrays` or `glDrawElements` calls, put as many vertices as possible between the `glBegin` and the `glEnd` calls.

16.5.3 Using Display Lists

You can often improve performance by storing frequently used commands in a display list. If you plan to redraw the same geometry multiple times, or if you have a set of state changes that need to be applied multiple times, consider using display lists. Display lists allow you to define the geometry and/or state changes once and execute them multiple times. Some graphics hardware may store display lists in dedicated memory or may store the data in an optimized form for rendering.

The biggest drawback of using display lists is data expansion. The display list contains an entire copy of all your data plus additional data for each command and for each list. As a result, tuning for display lists focuses mainly on reducing storage requirements. Performance improves if the data that is being traversed fits in the cache. Follow these rules to optimize display lists:

- Call `glDeleteLists` to delete display lists that are no longer needed. This frees storage space used by the deleted display lists and expedites the creation of new display lists.

- Avoid duplication of display lists. For example, if you have a scene with 100 spheres of different sizes and materials, generate one display list that is a unit sphere centered about the origin. Then reference the sphere many times, setting the appropriate material properties and transforms each time.

- Make the display lists as flat as possible, but be sure not to exceed the cache size. Avoid using an excessive hierarchy with many invocations to `glCallList`. Each `glCallList` invocation requires the OpenGL implementation to do some work (eg., a table lookup) to find the designated display list. A flat display list requires less memory and yields simpler and faster traversal. It also improves cache coherency.

  On the other hand, excessive flattening increases the size. For example, if you're drawing a car with four wheels, having a hierarchy with four pointers

from the body to one wheel is preferable to a flat structure with one body and four wheels.

- Avoid creating very small display lists. Very small lists may not perform well since there is some overhead when executing a list. Also, it is often inefficient to split primitive definitions across display lists.

- If appropriate, store state settings with geometry; it may improve performance.

  For example, suppose you want to apply a transformation to some geometric objects and then draw the result. If the geometric objects are to be transformed in the same way each time, it is better to store the matrix in the display list.

16.5.4 Balancing Polygon Size and Pixel Operations

The optimum size of polygons depends on the other operations going on in the pipeline:

- If the polygons are too large for the fill-rate to keep up with the rest of the pipeline, the application is fill-rate limited. Smaller polygons balance the pipeline and increase the polygon rate.

- If the polygons are too small for the rest of the pipeline to keep up with filling, then the application is transform limited. Larger and fewer polygons, or fewer vertices, balance the pipeline and increase the fill rate.

16.6 Rendering Images Efficiently

To improve performance when drawing pixel rectangles, follow these guidelines:

- Disable all per-fragment operations.

- Disable texturing and fog.

- Define images in the native hardware format so type conversion is not necessary.

- Know where the bottleneck is.

  Similar to polygon drawing, there can be a pixel-drawing bottleneck due to overload in host bandwidth, processing, or rasterizing. When all modes are off, the path is most likely limited by host bandwidth, and a wise choice of

host pixel format and type pays off tremendously. For this reason, using type `GL_UNSIGNED_BYTE`, for the image components is sometimes faster.

Zooming up pixels may create a raster bottleneck.

- A big pixel rectangle has a higher throughput (that is, pixels per second) than a small rectangle. Because the imaging pipeline is tuned to trade off a relatively large setup time with a high throughput, a large rectangle amortizes the setup cost over many pixels.

16.7 Tuning Animation

Tuning animation requires attention to some factors not relevant in other types of applications. This section discusses those factors.

16.7.1 Factors Contributing to Animation Speed

The smoothness of an animation depends on its frame rate. The more frames rendered per second, the smoother the motion appears.

Smooth animation also requires double buffering. In double buffering, one framebuffer holds the current frame, which is scanned out to the monitor by video hardware, while the rendering hardware is drawing into a second buffer that is not visible. When the new framebuffer is ready to be displayed, the system swaps the buffers. The system must wait until the next vertical retrace period between raster scans to swap the buffers, so that each raster scan displays an entire stable frame, rather than parts of two or more frames.

Frame rates must be integral multiples of the screen refresh time, which is 16.7 msec (milliseconds) for a 60-Hz monitor. If the draw time for a frame is slightly longer than the time for *n* raster scans, the system waits until the *n+1st* vertical retrace before swapping buffers and allowing drawing to continue, so the total frame time is (*n*+1)*16.7 msec.

To summarize: A change in the time spent rendering a frame has no visible effect unless it changes the total time to a different integer multiple of the screen refresh time.

If you want an observable performance increase, you must reduce the rendering time enough to take a smaller number of 16.7 msec raster scans. Alternatively, if performance is acceptable, you can add work without reducing performance, as long as the rendering time does not exceed the current multiple of the raster scan time.

To help monitor timing improvements, turn off double buffering. If you don't, it's difficult to know if you're near a 16.7 msec boundary.

16.7.2 Optimizing Frame Rate Performance

The most important aid for optimizing frame rate performance is taking timing measurements in single-buffer mode only. For more detailed information, see "Taking Timing Measurements".

In addition, follow these guidelines to optimize frame rate performance:

- Reduce drawing time to a lower multiple of the screen refresh time.

  This is the only way to produce an observable performance increase.

- Perform non-graphics computation after swapping buffers.

  A program is free to do non-graphics computation during the wait cycle between vertical retraces. Therefore, the procedure for rendering a frame is: call swap buffers immediately after sending the last graphics call for the current frame, perform computation needed for the next frame, then execute OpenGL calls for the next frame.

- Do non-drawing work after a screen clear.

  Clearing a full screen takes time. If you make additional drawing calls immediately after a screen clear, you may fill up the graphics pipeline and force the program to stall. Instead, do some non-drawing work after the clear.

16.8 Taking Timing Measurements

Timing, or benchmarking, parts of your program is an important part of tuning. It helps you determine which changes to your code have a noticeable effect on the speed of your application.

To achieve performance that is demonstrably close to the best the hardware can achieve, you can first follow the more general tuning tips provided here, but you then need to apply a rigorous and systematic analysis.

16.8.1 Benchmarking Basics

A detailed analysis involves examining what your program is asking the system to do and then calculating how long that should take, based on the known performance characteristics of the hardware. Compare this calculation of expected performance with the performance actually observed and continue to apply the tuning techniques until the two match more closely. At this point, you have a detailed accounting of how your program spends its time, and you are in a strong position both to tune further and to make appropriate decisions considering the speed-versus-quality trade-off.

The following parameters determine the performance of most applications:

- Total number of polygons in a frame
- Transform rate for the given polygon type and mode settings
- Number of pixels filled
- Fill rate for the given mode settings
- Time of color and depth buffer clear
- Time of buffer swap
- Time of application overhead
- Number of attribute changes and time per change

16.8.2 Achieving Accurate Timing Measurements

Consider these guidelines to get accurate timing measurements:

- Take measurements on a quiet system. Verify that no unusual activity is taking place on your system while you take timing measurements. Terminate other applications. For example, don't have a clock or a network application running while you are benchmarking.
- Choose timing trials that are not limited by the clock resolution.

  Use a high-resolution clock and make measurements over a period of time that's at least one hundred times the clock resolution. A good rule of thumb is to benchmark something that takes at least two seconds so that the uncertainty contributed by the clock reading is less than one percent of the total error. To measure something that's faster, write a loop to execute the test code repeatedly.
- Benchmark static frames.

  Verify that the code you are timing behaves identically for each frame of a given timing trial. If the scene changes, the current bottleneck in the graphics pipeline may change, making your timing measurements meaningless. For example, if you are benchmarking the drawing of a rotating airplane, choose a single frame and draw it repeatedly, instead of letting the airplane rotate. Once a single frame has been analyzed and tuned, look at frames that stress the graphics pipeline in different ways, then analyze and tune them individually.

- Compare multiple trials.

  Run your program multiple times and try to understand variance in the trials. Variance may be due to other programs running, system activity, prior memory placement, or other factors.

- Call `glFinish` before reading the clock at the start and at the end of the time trial.

  This is important if you are using a machine with hardware acceleration because the graphics commands are put into a hardware queue in the graphics subsystem, to be processed as soon as the graphics pipeline is ready. The CPU can immediately do other work, including issuing more graphics commands until the queue fills up.

  When benchmarking a piece of graphics code, you must include in your measurements the time it takes to process all the work left in the queue after the last graphics call. Call `glFinish` at the end of your timing trial, just before sampling the clock. Also call `glFinish` before sampling the clock and starting the trial, to ensure no graphics calls remain in the graphics queue ahead of the process you are timing.

16.8.3 Achieving Accurate Benchmarking Results

To benchmark performance for a particular code fragment, follow these steps:

- Determine how many polygons are being drawn and estimate how many pixels they cover on the screen. Have your program count the polygons when you read in the database. To determine the number of pixels filled, start by making a visual estimate. Be sure to include surfaces that are hidden behind other surfaces, and notice whether or not backface elimination is enabled. For greater accuracy, use feedback mode and calculate the actual number of pixels filled or use the stencil buffer technique described in Section 14.3.

- Determine the transform and fill rates on the target system for the mode settings you are using. Refer to the product literature for the target system to determine some transform and fill rates. Determine others by writing and running small benchmarks.

- Divide the number of polygons drawn by the transform rate to get the time spent on per-polygon operations.

- Divide the number of pixels filled by the fill rate to get the time spent on per-pixel operations.

- Measure the time spent in the application. To determine time spent executing instructions in the application, stub out the OpenGL calls and benchmark your application.

This process takes some effort to complete. In practice, it's best to make a quick start by making some assumptions, then refine your understanding as you tune and experiment. Ultimately, you need to experiment with different rendering techniques and do repeated benchmarks, especially when the unexpected happens.

17 List of Demo Programs

This list shows the demonstration programs available on the Programming with OpenGL: Advanced Rendering web site at:

`http://www.sgi.com/Technology/OpenGL/advanced_sig97.html`

The programs are grouped by the sections in which they're discussed. Each line gives a short description of the program.

Modelling

- tvertex.c - show problems caused by t-vertices
- quad_decomp.c - shows example of quadrilateral decomposition
- tess.c - shows examples of sphere tessellation
- cap.c - shows how to cap the region exposed by a clipping plane
- csg.c - shows how to render CSG solids with the stencil buffer

Geometry and Transformations

- depth.c - compare screen and eye space z
- decal.c - shows how to decal coplanar polygons with the stencil buffer
- hiddenline.c - shows how to render wireframe objects with hidden lines
- stereo.c - shows how to generate stereo image pairs
- tile.c - shows how to tile images
- raster.c - shows how to move the current raster position off-screen

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
- projtex.c - shows how to do spotlight illumination using projective textures
- cyl_billboard.c - shows how to do cylindrical billboards
- sph_billboard.c - shows how to do spherical billboards
- warp.c - shows how to warp images with textures
- noise.c - shows how to make a filtered noise function
- spectral.c - shows how to make a spectral function from filtered noise
- spotnoise.c - shows how to use spot noise
- tex3dsolid.c - renders a solid image with a 3d texture
- tex3dfunc.c - creates a 2d texture that varies with r value

Blending

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

Antialiasing

- lineaa.c - shows how to draw antialiased lines
- texaa.c - shows how to antialias with texture
- accumaa.c - shows how to antialias a scene with the accumulation buffer

Lighting

- envphong.c - shows how to draw phong highlights with environment mapping

- lightmap2d.c - shows how to do 2D texture lightmaps
- lightmap3d.c - shows how to do 3D texture lightmaps
- bumpmap.c - shows how to bumpmap with texture

Scene Realism

- motionblur.c - shows how to do motion blur with the accumulation buffer
- field.c - shows how to achieve depth of field effects with the accumulation buffer
- genspheremap.c - shows how to generate sphere maps
- mirror.c - shows how to do planar mirror reflections
- 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
- softshadow.c - shows how to do soft shadows with the accumulation buffer by jittering light sources
- softshadow2.c - shows how to do soft shadows by creating lighting textures with the accumulation buffer

Transparency

- screendoor.c - shows how to do screen-door transparency
- alphablend.c - shows how to do transparency with alpha blending

Natural Phenomena

- smoke.c - shows how to render smoke
- smoke3d.c - shows how to render 3D smoke using volumetric techniques
- cloud.c - shows how to render a cloud layer
- cloud3d.c - shows how to render a 3D cloud using volumetric techniques
- fire.c - shows how to render fire using movie loops
- water.c - shows an example water rendering technique

- lightpoint.c - shows how to render point light sources

Image Processing

- convolve.c - shows how to convolve with the accumulation buffer
- cmatrix - shows how to modify colors with a color matrix

Volume Visualization with Texture

- vol2dtex.c - volume visualization with 2D textures
- vol3dtex.c - volume visualization with 3D textures

Using the Stencil Buffer

- dissolve.c - shows how to do dissolves with the stencil buffer
- zcomposite.c - shows how to composite depth-buffered images with the stencil buffer

Line Rendering Techniques

- 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

18 Equation Appendix

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

18.1 Projection Matrices

18.1.1 Perspective Projection

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

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

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

18.1.2 Orthographic Projection

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

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

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

18.2 Lighting Equations

18.2.1 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 directional, the attenuation factor is 1.

18.2.2 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.

- $(\max\{v \cdot d, 0\})^{GL\_SPOT\_EXPONENT}$ 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 $(\max\{\hat{v} \cdot \hat{d}, 0\})$ where $\hat{v}$ and $\hat{d}$ 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.

18.2.3 Ambient Term

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

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

18.2.4 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:

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

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.

18.2.5 Specular Term

The specular term also depends on whether light falls directly on the vertex. If $\vec{l} \cdot \vec{n}$ 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:

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

However, if $\vec{l} \cdot \vec{n} = 0$, the specular term is 0.

18.2.6 Putting It All Together

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

$$\begin{aligned}
\text{vertex color} \quad = \quad & \text{emission}_{material} + \\
& \text{ambient}_{lightmodel} * \text{ambient}_{material} + \\
& \sum_{i=0}^{n-1} \left(\frac{1}{k_c + k_l d + k_q d^2} \right) (\text{spotlight effect})_i \\
& (\text{ambient}_{light} * \text{ambient}_{material} + \\
& (\max\{l \cdot n, 0\}) * \text{diffuse}_{light} * \text{diffuse}_{material} + \\
& (\max\{s \cdot n, 0\})^{shininess} * \text{specular}_{light} * \text{specular}_{material})_i
\end{aligned}$$

19 References

References

[1] J. Airey, B. Cabral, and M. Peercy. Explanation of bump mapping with texture. Personal Communication, 1997.

[2] K. Akeley. The hidden charms of z-buffer. *Iris Universe*, (11):31–37, 1990.

[3] K. Akeley. OpenGL philosophy and the philosopher's drinking song. Personal Communication, 1996.

[4] Y. Attarwala. Rendering hidden lines. *Iris Universe*, Fall:39, 1988.

[5] Y. Attarwala and M. Kong. Picking from the picked few. *Iris Universe*, Summer:40–41, 1989.

[6] James F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12, pages 286–292, August 1978.

[7] OpenGL Architecture Review Board. *OpenGL Reference Manual*. Addison-Wesley, Menlo Park, 1992.

[8] B. Cabral and C. Leedom. Highly parallel vector visualization using line integral convolution. In *Proceedings of the Seventh Siam Conference On Parallel Processing for Scientific Computing*, volume 7, pages 803–807, 1995.

[9] The VRML Consortium. The virtual reality modeling language specification. web site, August 1996. http://vag.vrml.org.

[10] J. D. Cutnell and K. W. Johnson. *Physics*. John Wiley & Sons, 1989.

[11] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 65–74, August 1988.

[12] Tom Duff. Compositing 3-D rendered images. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 41–44, July 1985.

[13] David Ebert, Kent Musgrave, Darwyn Peachey, Ken Perlin, and Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, October 1994. ISBN 0-12-228760-6.

[14] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Company, 1990.

[15] Alain Fournier and William T. Reeves. A simple model of ocean waves. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 75–84, August 1986.

[16] Geoffrey Y. Gardner. Visual simulation of clouds. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 297–303, July 1985.

[17] Jack Goldfeather, Jeff P. M. Hultquist, and Henry Fuchs. Fast constructive-solid geometry display in the Pixel-Powers graphics system. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 107–116, August 1986.

[18] P. Haeberli. Matrix operations for image processing. web site, November 1993. http://www.sgi.com/grafica/matrix/index.html.

[19] P. Haeberli and D. Voorhies. Image processing by linear interpolation and extrapolation. *Iris Universe*, (28):8–9, 1994.

[20] Paul Haeberli and Mark Segal. Texture mapping as a fundamental drawing primitive. In Michael F. Cohen, Claude Puech, and Francois Sillion, editors, *Fourth Eurographics Workshop on Rendering*, pages 259–266. Eurographics, June 1993. held in Paris, France, 14–16 June 1993.

[21] Paul E. Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 309–318, August 1990.

[22] Peter M. Hall and Alan H. Watt. Rapid volume rendering using a boundary-fill guided ray cast algorithm. In N. M. Patrikalakis, editor, *Scientific Visualization of Physical Phenomena (Proceedings of CG International '91)*, pages 235–249. Springer-Verlag, 1991.

[23] Roy Hall. *Illumination and Color in Computer Generated Imagery*. Springer-Verlag, New York, 1989. includes C code for radiosity algorithms.

[24] Paul S. Heckbert and Michael Herf. Fast soft shadows. In *Visual Proceedings, SIGGRAPH 96*, page 145. ACM Press, 1996. ISBN 0-89791-784-7.

[25] Paul S. Heckbert and Michael Herf. Shadow generation algorithms. web site, April 1997. http://www.cs.cmu.edu/ ph/shadow.html.

[26] T. Heidmann. Real shadows real time. *Iris Universe*, (18):28–31, 1991.

[27] Russ Herrell, Joe Baldwin, and Chris Wilcox. High quality polygon edging. *IEEE Computer Graphics and Applications*, 15(4):68–74, July 1995.

[28] Michael Kass and Gavin Miller. Rapid, stable fluid dynamics for computer graphics. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 49–57, August 1990.

[29] John-Peter Lewis. Algorithms for solid noise synthesis. In Jeffrey Lane, editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 263–270, July 1989.

[30] Don P. Mitchell and Arun N. Netravali. Reconstruction filters in computer graphics. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 221–228, August 1988.

[31] H. R. Myler and A. R. Weeks. *The Pocket Handbook of Image Processing Algorithms in C.* University of Central Florida Department of Electrical & Computer Engineering, 1993.

[32] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison-Wesley, Menlo Park, 1993.

[33] Tomoyuki Nishita and Eihachiro Nakamae. Method of displaying optical effects within water using accumulation buffer. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 373–381. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[34] Darwyn R. Peachey. Modeling waves and surf. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 65–74, August 1986.

[35] M. Peercy. Explanation of sphere mapping. Personal Communication, 1997.

[36] Mark Peercy, John Airey, and Brian Cabral. Efficient bump mapping hardware. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, 1997.

[37] Bui-T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

[38] Thomas Porter and Tom Duff. Compositing digital images. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 253–259, July 1984.

[39] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 283–291, July 1987.

[40] P. Rustagi. Silhouette line display from shaded models. *Iris Universe*, Fall:42–44, 1989.

[41] P. Rustagi. Image roaming with the help of tiling and memory-mapped files. *Iris Universe*, (15):12–13, 1991.

[42] John Schlag. *Fast Embossing Effects on Raster Image Data*. Academic Press, Cambridge, 1994.

[43] M. Schulman. Rotation alternatives. *Iris Universe*, Spring:39, 1989.

[44] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul E. Haeberli. Fast shadows and lighting effects using texture mapping. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 249–252, July 1992.

[45] M. Teschner. Texture mapping: New dimensions in scientific and technical visualization. *Iris Universe*, (29):8–11, 1994.

[46] T. Tessman. Casting shadows on flat surfaces. *Iris Universe*, Winter:16, 1989.

[47] Jarke J. van Wijk. Spot noise-texture synthesis for data visualization. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 309–318, July 1991.

[48] Douglas Voorhies and Jim Foran. Reflection vector shading hardware. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 163–166. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[49] Mark Watt. Light-water interaction using backward beam tracing. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 377–385, August 1990.

[50] T. F. Wiegand. Interactive rendering of csg models. In *Computer Graphics Forum*, volume 15, pages 249–261, 1996.

[51] Lance Williams. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, volume 17, pages 1–11, July 1983.

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

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notie is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or permission.

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-

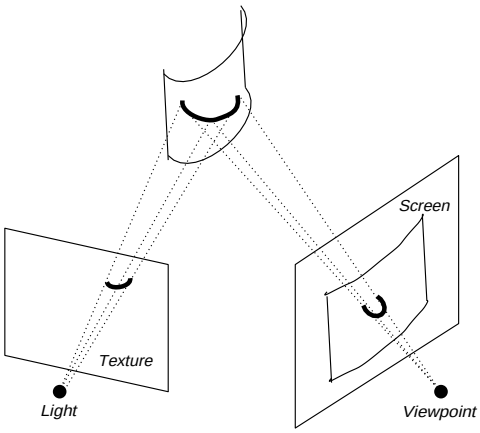


Figure 1. Viewing a projected texture.

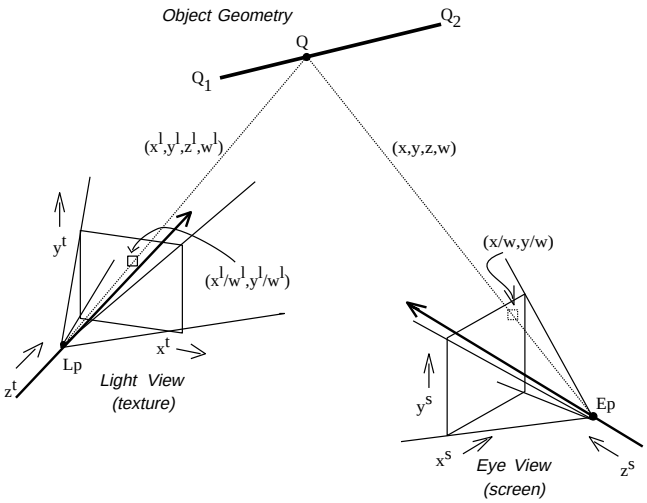


Figure 2. Object geometry in the light and clip coordinate systems.

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)*

© SGI, August 4, 1995

© SGI, August 4, 1995

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.

© SGI, August 4, 1995

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.

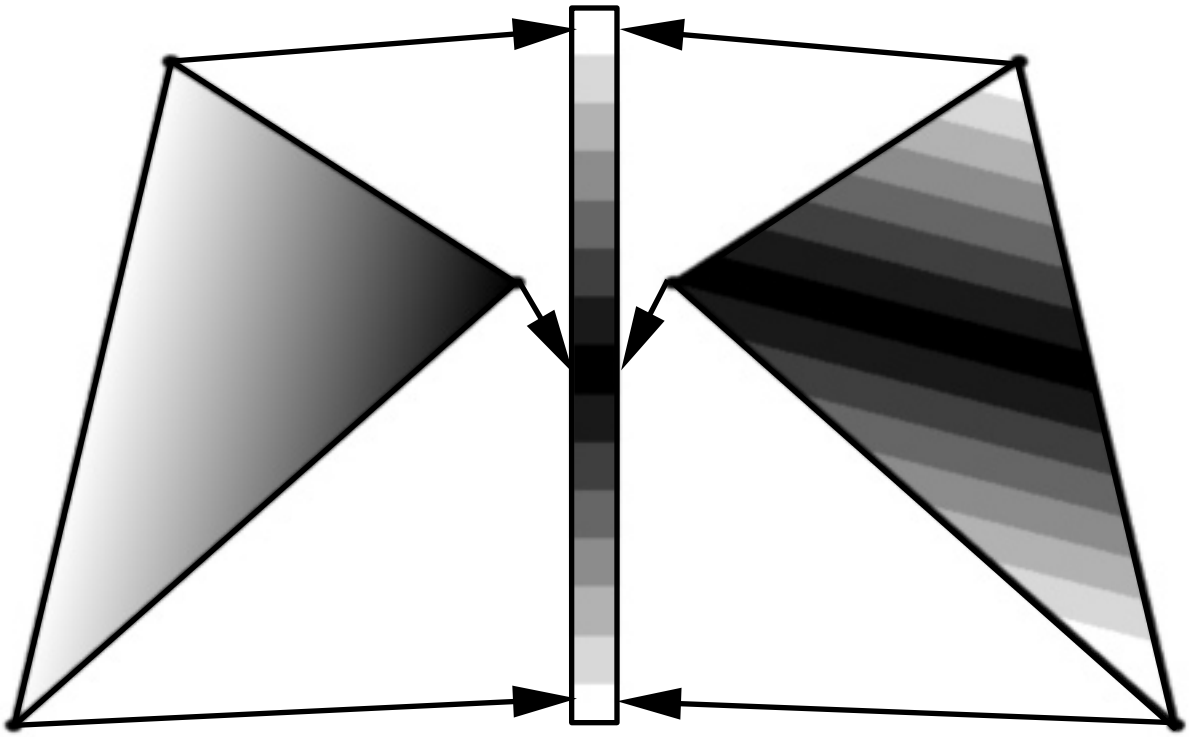


Figure 1: *Color coding with RGB interpolation (left) and texture mapping (right).*

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.

© SGI, August 4, 1995

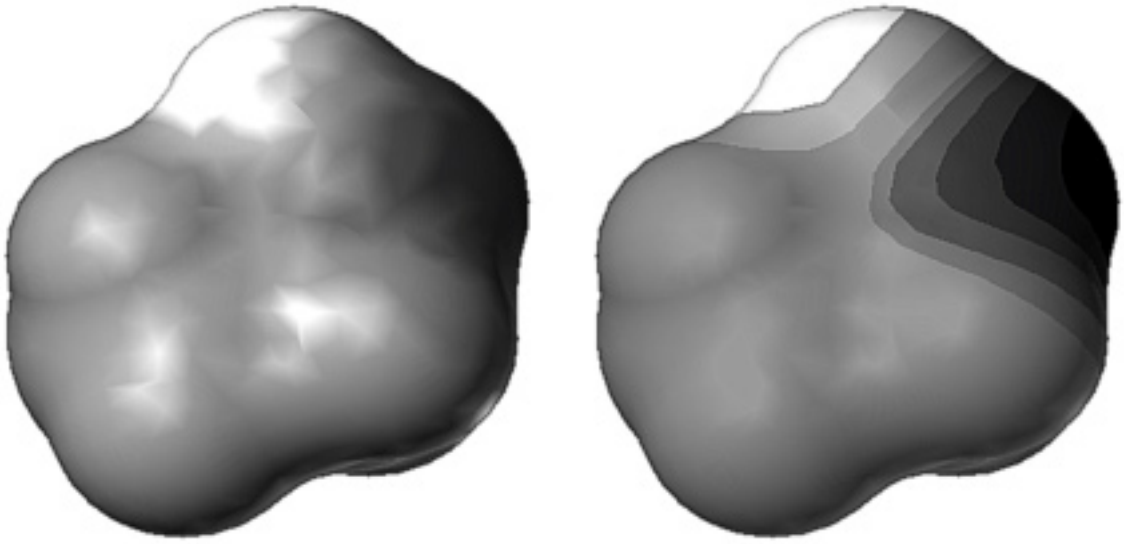


Figure 2: *Electrostatic potential coded on the solvent accessible surface of ethanol.*

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.

© SGI, August 4, 1995

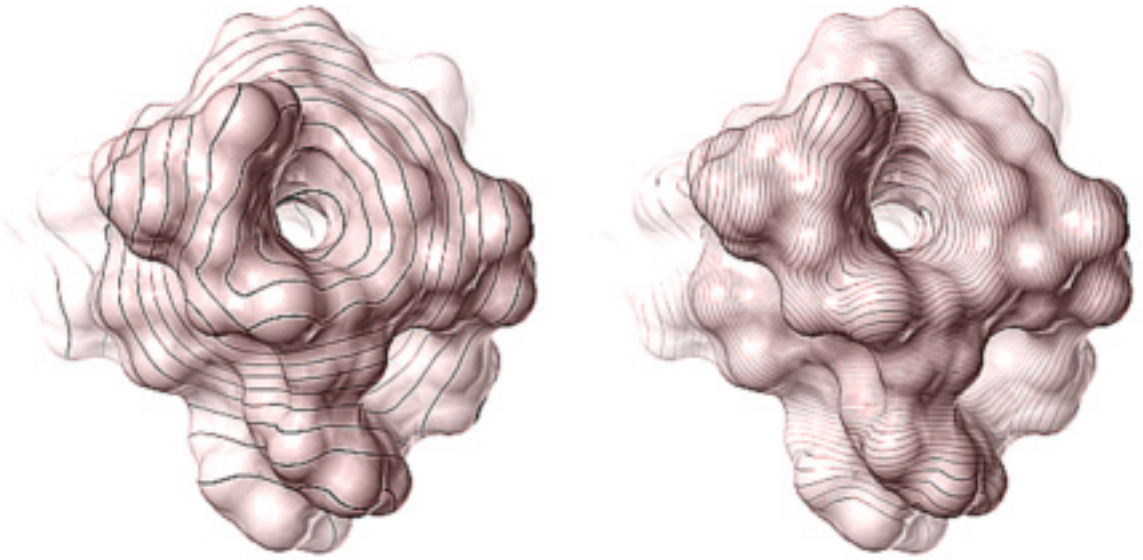


Figure 3: *Isocontour on a molecular surface with different scaling in texture space.*

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.

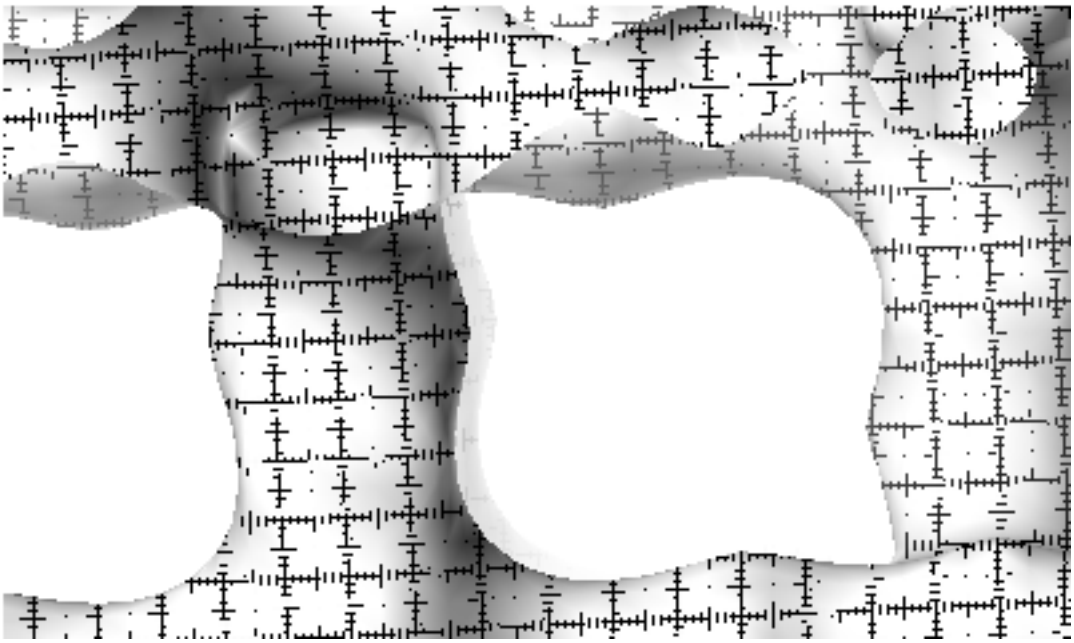


Figure 4: *Display of metrics on a Zeolithe's molecular surface with a 2−D texture.*

© SGI, August 4, 1995

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.

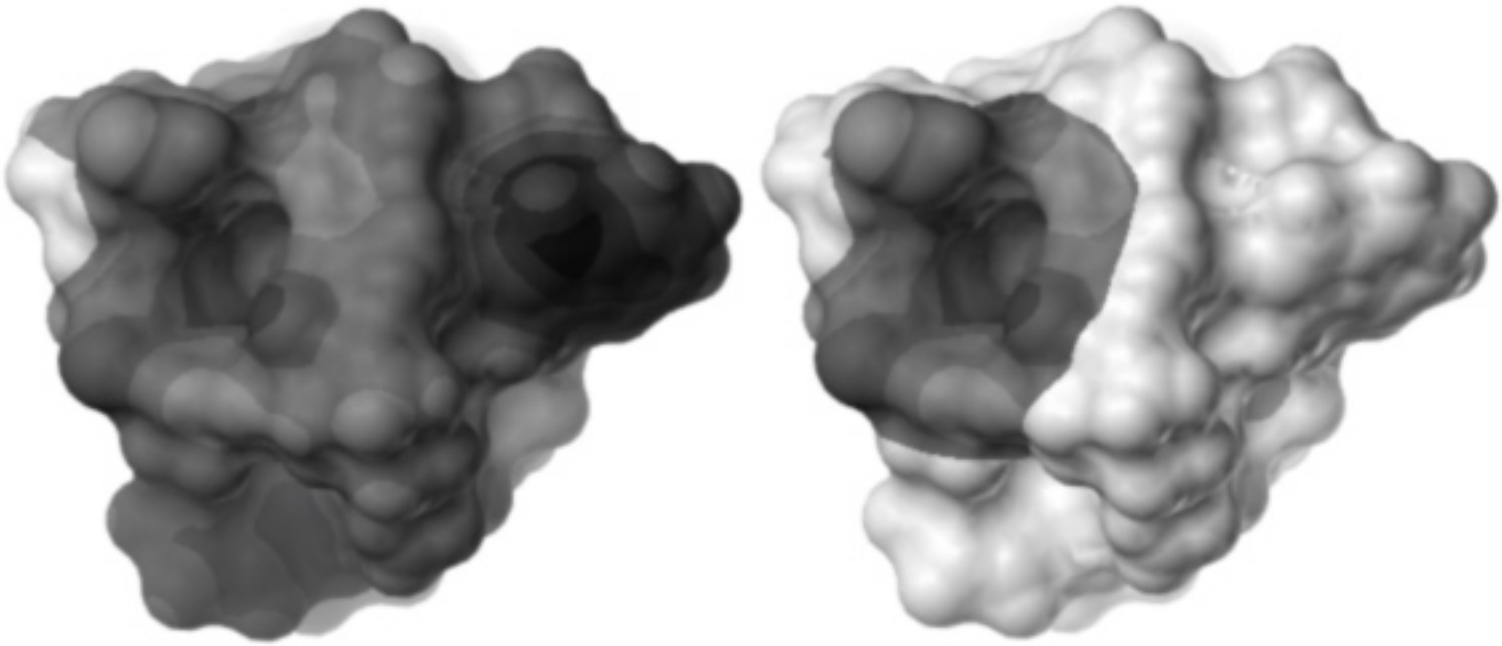


Figure 5: *Solvent accessible surface of Gramicidine A, showing the ESP filtered with the MLP.*

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.

© SGI, August 4, 1995

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.

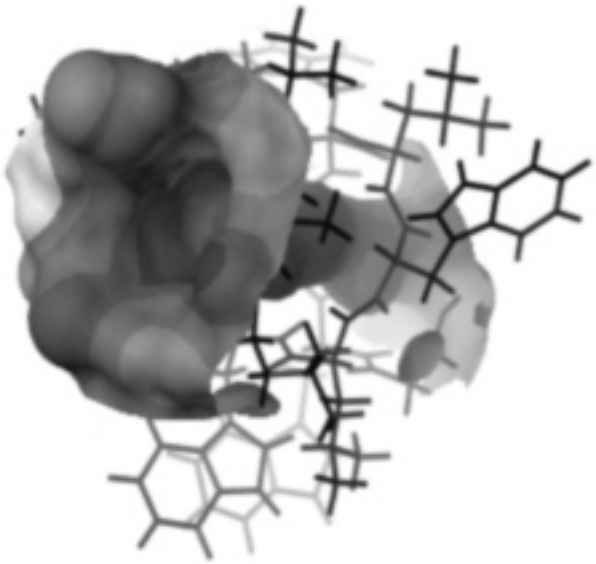


Figure 6: *Clipping of the solvent accessible surface of Gramicidine A according to the MLP.*

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!

© SGI, August 4, 1995

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.

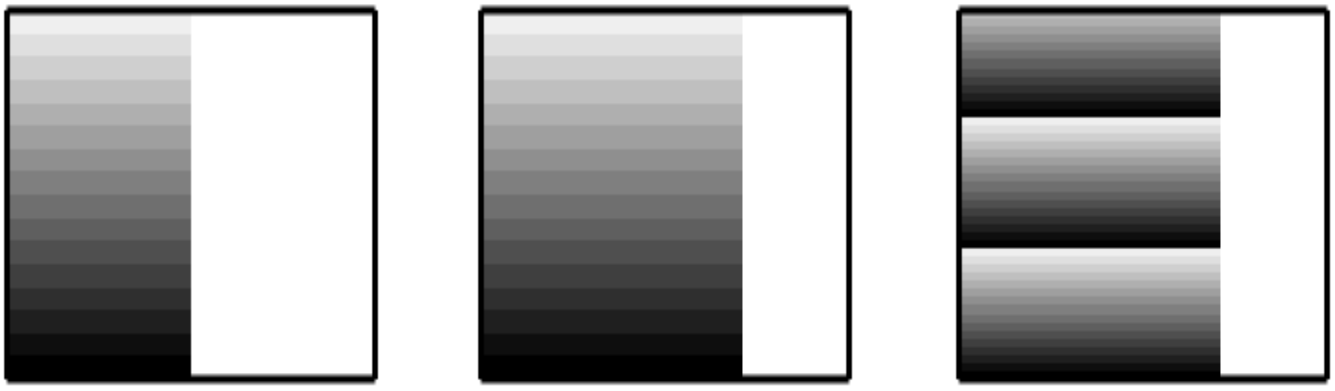


Figure 7: *Example of a 2−D texture used for information filtering, with different transformations applied: original texture (left), translation in s coordinates to adjust filter threshold (middle) and scaling along in t coordinates to change meaning of the texture colors (right).*

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();
```

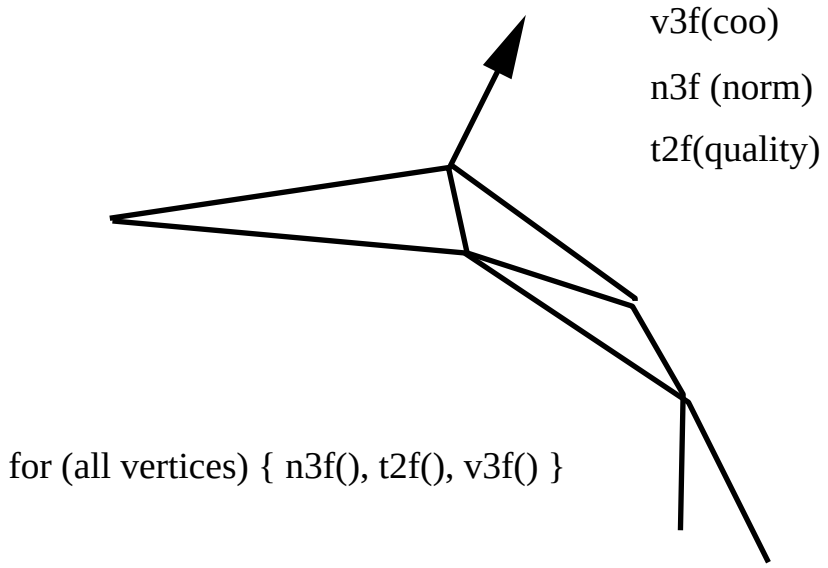


Figure 8: *Schematic representation of the* `drawTexturedSurface()` *routine.*

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.

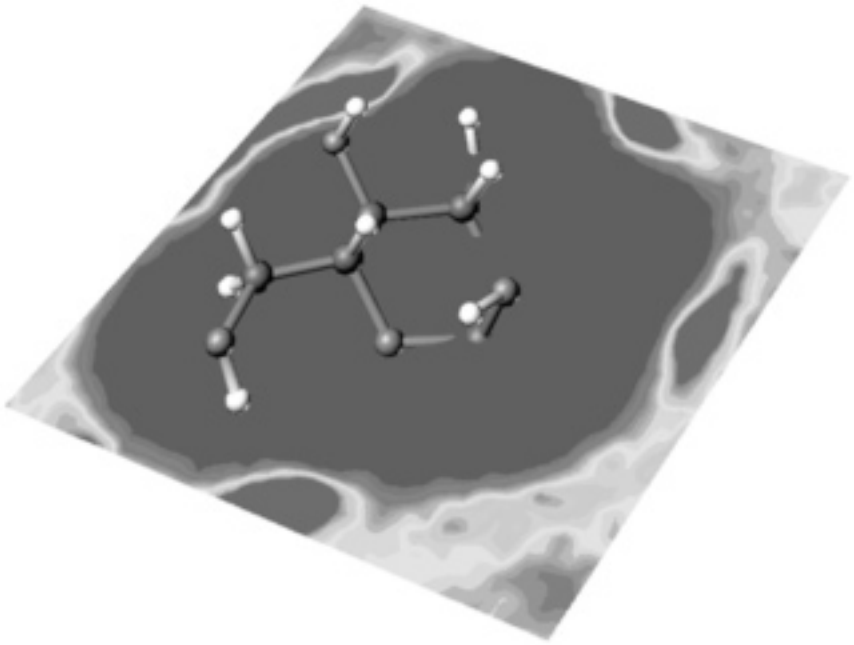


Figure 9: *Slice plane through the water density surrounding a sugar molecule.*

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.

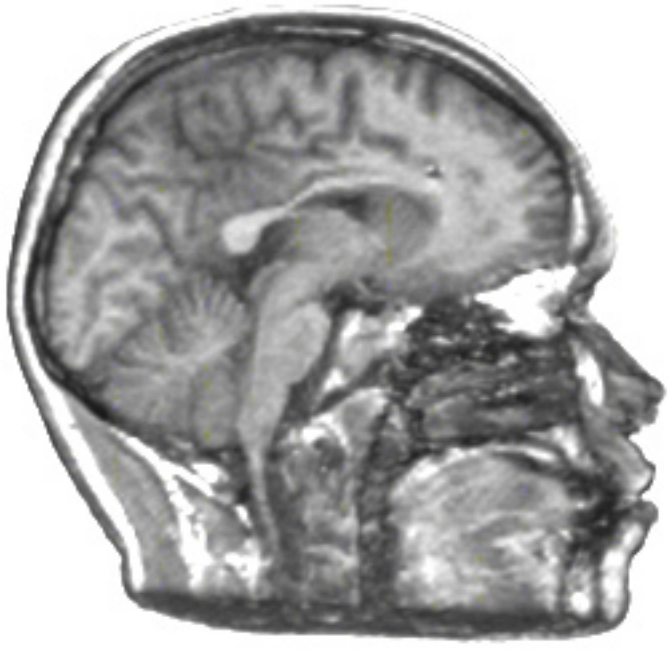


Figure 10: *Volume rendering of MRI data using a stack of screen−parallel sectioning planes, which is cut in half to reveal detail in the inner part of the object.*

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.

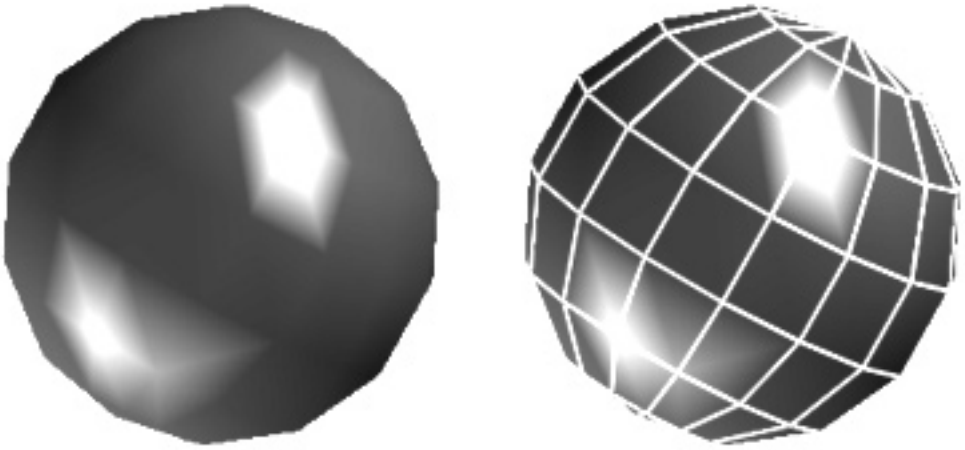


Figure 11: *Gouroud shading artifacts on a moderately tessellated sphere.*

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.

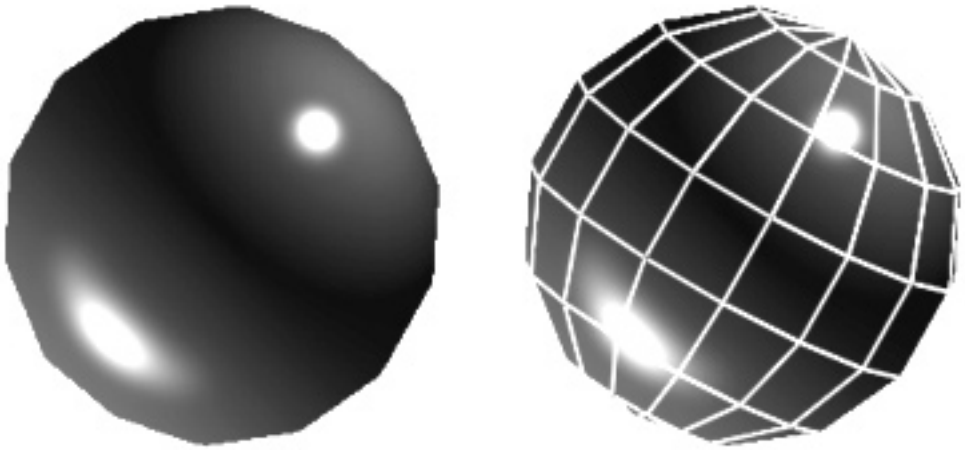


Figure 12: *Phong shaded sphere using surface normals as a lookup for the texture coordinate.*

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.

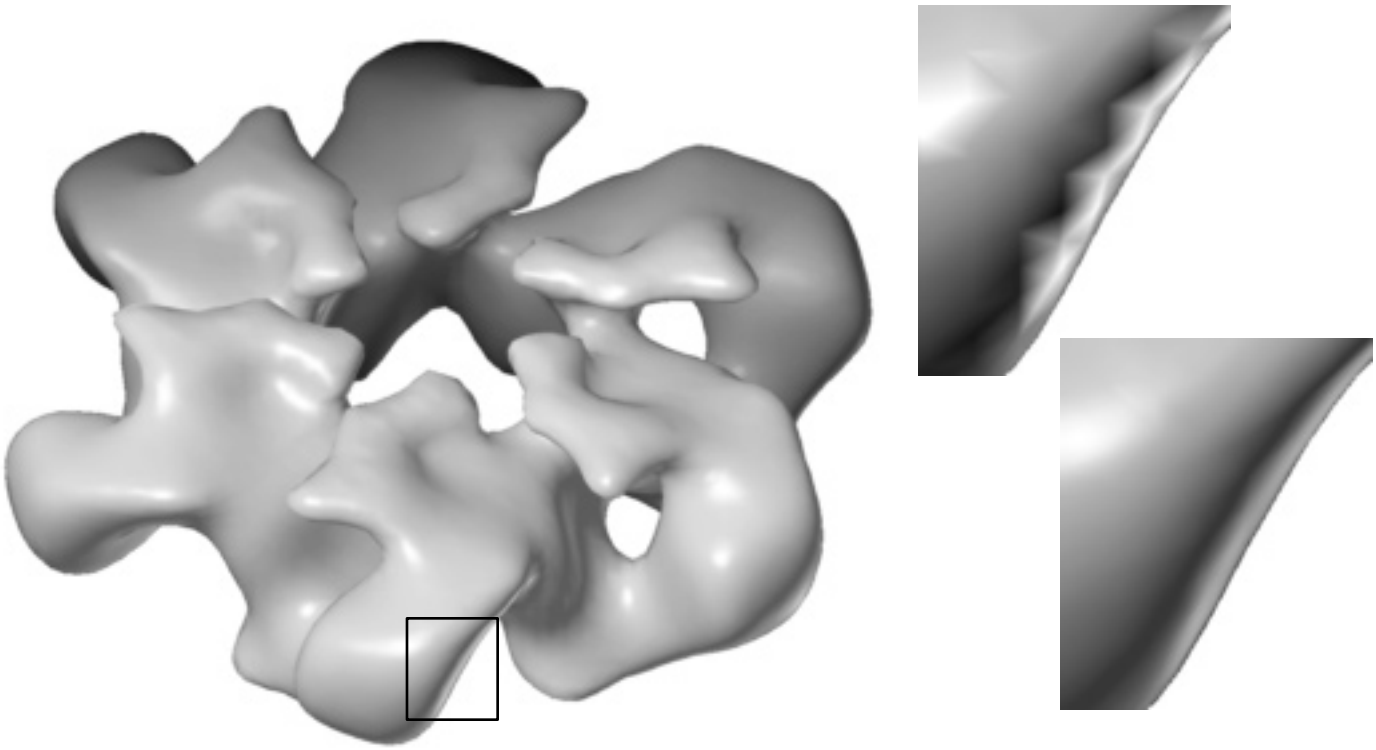


Figure 13: *Application of the texture mapped Phong shading to a complex surface representing a biomolecular structure. The closeups demonstrate the difference between Gouraud shading (above right) and Phong shading (below right) when implemented using texture mapping*

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();
```

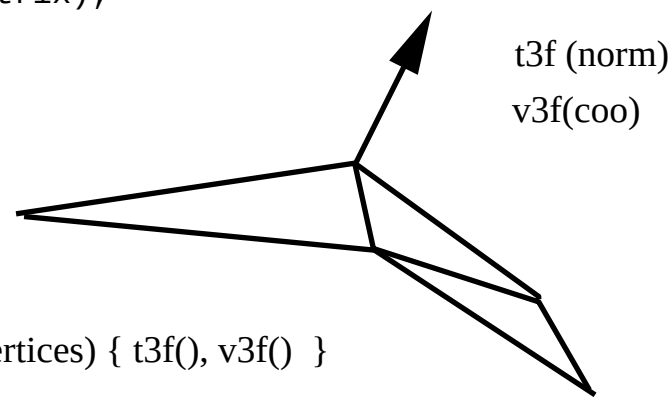


Figure 15: *Schematic representation of the* `drawTexPhongSurface()` *routine.*

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

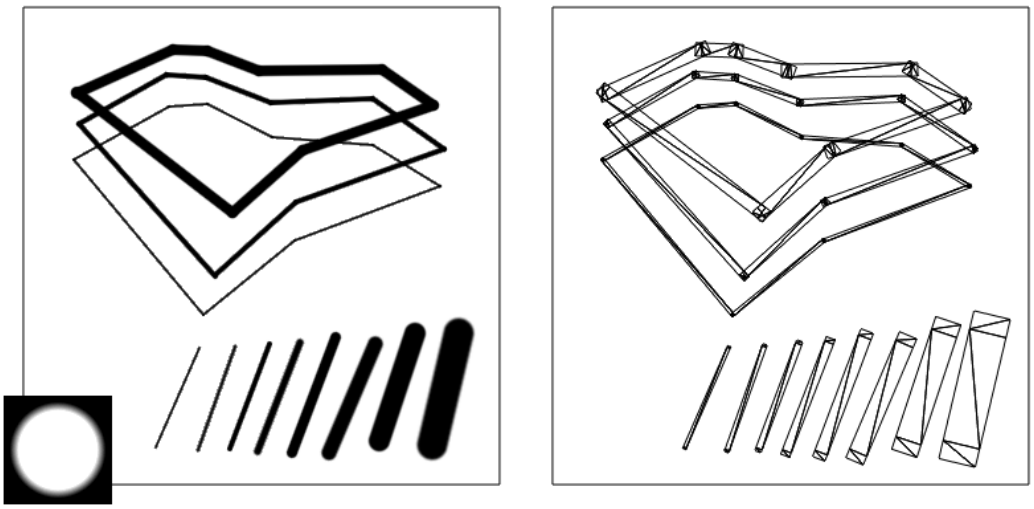


Figure 1. Anti-aliased line segments.

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.

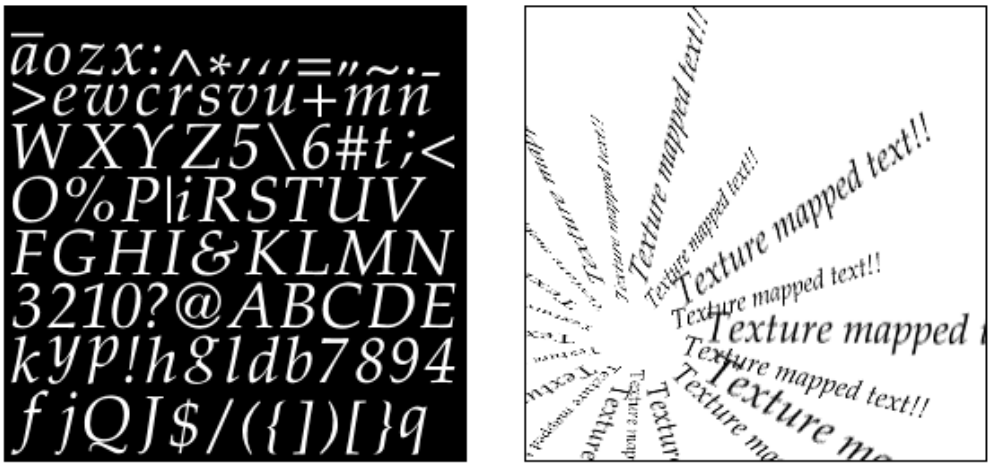


Figure 3. Anti-aliased text.

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-

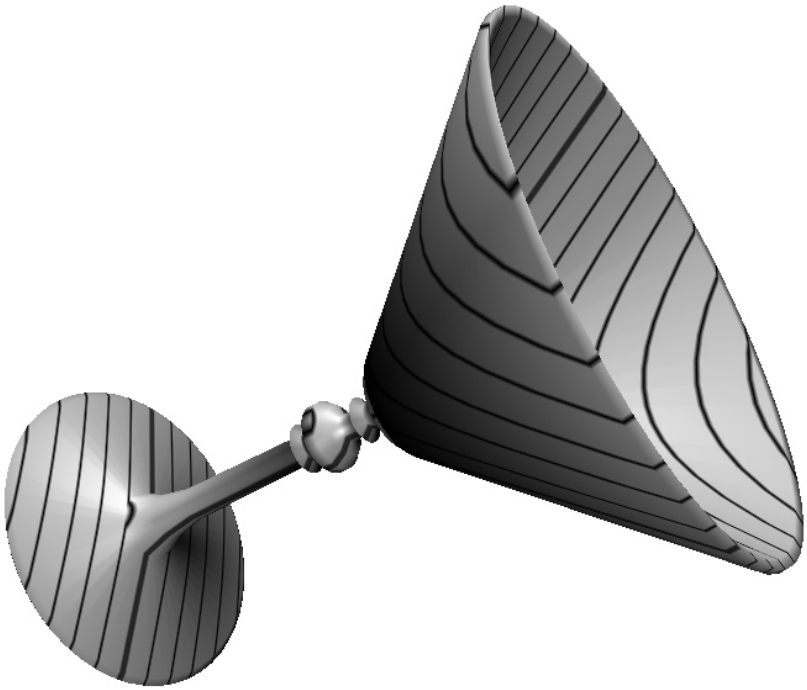


Figure 4. Contouring showing distance from a plane.

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

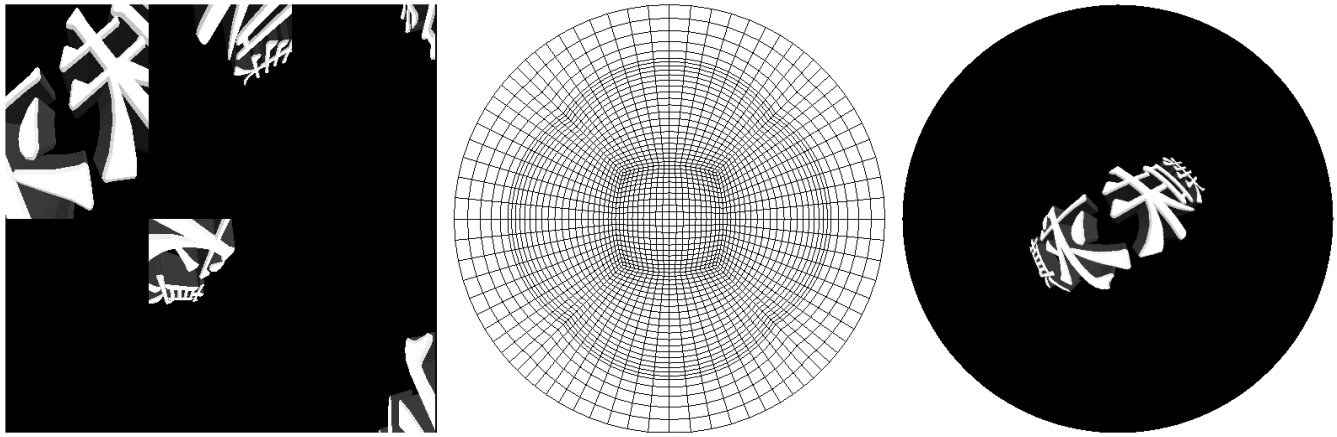


Figure 5. 360 Degree fisheye projection.

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

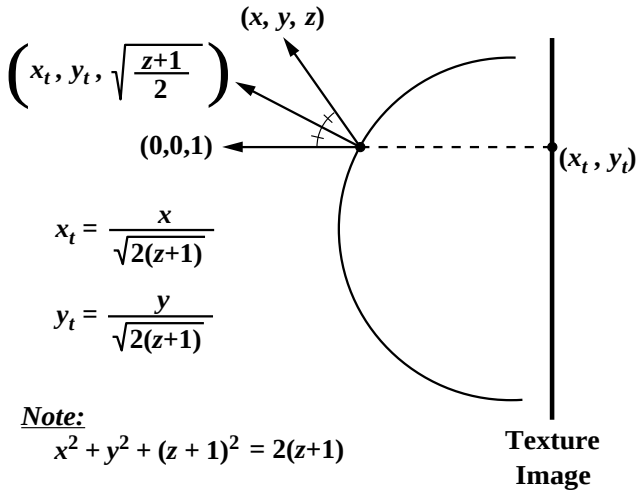


Figure 6. Spherical reflection geometry.

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].

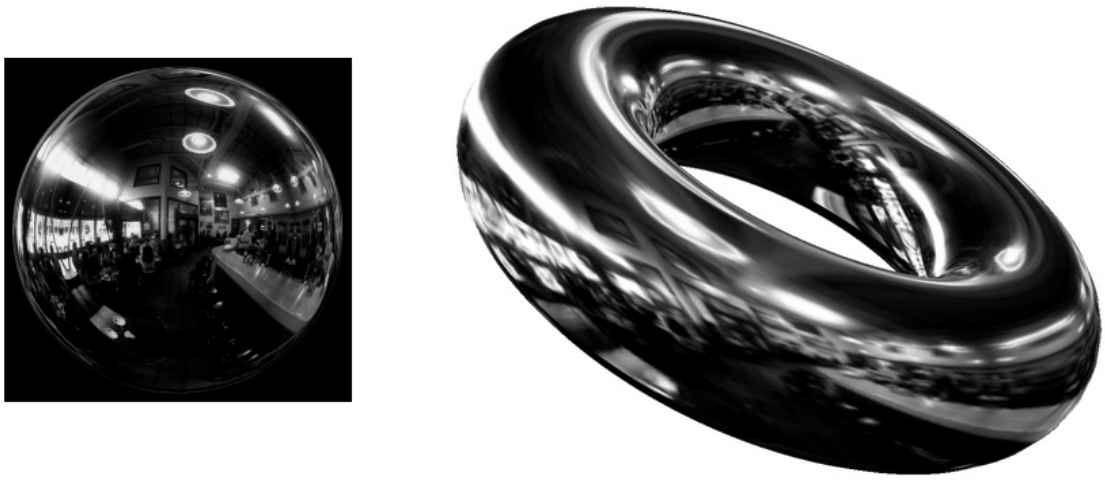


Figure 7. Environment mapping.

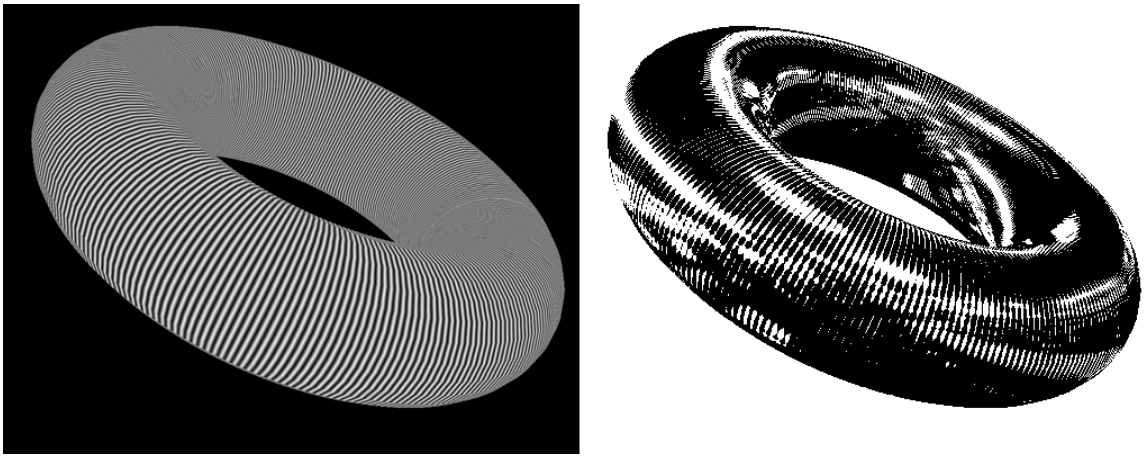


Figure 8. 3D halftoning.

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.

Simulating Soft Shadows with Graphics Hardware

Paul S. Heckbert and Michael Herf

January 15, 1997
CMU-CS-97-104

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

email: ph@cs.cmu.edu, herf+@cmu.edu
World Wide Web: http://www.cs.cmu.edu/∼ph

This paper was written in April 1996. An abbreviated version appeared in [Michael Herf and Paul S. Heckbert, Fast Soft Shadows, *Visual Proceedings, SIGGRAPH 96*, Aug. 1996, p. 145].

Abstract

This paper describes an algorithm for simulating soft shadows at interactive rates using graphics hardware. On current graphics workstations, the technique can calculate the soft shadows cast by moving, complex objects onto multiple planar surfaces in about a second. In a static, diffuse scene, these high quality shadows can then be displayed at 30 Hz, independent of the number and size of the light sources.

For a diffuse scene, the method precomputes a *radiance texture* that captures the shadows and other brightness variations on each polygon. The texture for each polygon is computed by creating registered projections of the scene onto the polygon from multiple sample points on each light source, and averaging the resulting hard shadow images to compute a soft shadow image. After this precomputation, soft shadows in a static scene can be displayed in real-time with simple texture mapping of the radiance textures. All pixel operations employed by the algorithm are supported in hardware by existing graphics workstations. The technique can be generalized for the simulation of shadows on specular surfaces.

This work was supported by NSF Young Investigator award CCR-9357763. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF or the U.S. government.

Keywords: penumbra, texture mapping, graphics workstation, interaction, real-time, SGI Reality Engine.

1 Introduction

Shadows are both an important visual cue for the perception of spatial relationships and an essential component of realistic images. Shadows differ according to the type of light source causing them: point light sources yield hard shadows, while linear and area (also known as *extended*) light sources generally yield soft shadows with an umbra (fully shadowed region) and penumbra (partially shadowed region).

The real world contains mostly soft shadows due to the finite size of sky light, the sun, and light bulbs, yet most computer graphics rendering software simulates only hard shadows, if it simulates shadows at all. Excessive sharpness of shadow edges is often a telltale sign that a picture is computer generated.

Shadows are even less commonly simulated with hardware rendering. Current graphics workstations, such as Silicon Graphics (SGI) and Hewlett Packard (HP) machines, provide z-buffer hardware that supports real-time rendering of fairly complex scenes. Such machines are wonderful tools for computer aided design and visualization. Shadows are seldom simulated on such machines, however, because existing algorithms are not general enough, or they require too much time or memory. The shadow algorithms most suitable for interaction on graphics workstations have a cost per frame proportional to the number of point light sources. While such algorithms are practical for one or two light sources, they are impractical for a large number of sources or the approximation of extended sources.

We present here a new algorithm that computes the soft shadows due to extended light sources. The algorithm exploits graphics hardware for fast projective (perspective) transformation, clipping, scan conversion, texture mapping, visibility testing, and image averaging. The hardware is used both to compute the shading on the surfaces and to display it, using texture mapping. For diffuse scenes, the shading is computed in a preprocessing step whose cost is proportional to the number of light source samples, but while the scene is static, it can be redisplayed in time independent of the number of light sources. The method is also useful for simulating the hard shadows due to a large number of point sources. The memory requirements of the algorithm are also independent of the number of light source samples.

1.1 The Idea

For diffuse scenes, our method works by precomputing, for each polygon in the scene, a *radiance texture* [12,14] that records the color (outgoing radiance) at each point in the polygon. In a diffuse scene, the radiance at each surface point is view independent, so it can be precomputed and re-used until the scene geometry changes. This radiance texture is analogous to the mesh of radiosity values computed in a radiosity algorithm. Unlike a radiosity algorithm, however, our algorithm can compute this texture almost entirely in hardware.

The key idea is to use graphics hardware to determine visibility and calculate shading, that is, to determine which portions of a surface are occluded with respect to a given extended light source, and how brightly they are lit. In order to simulate extended light sources, we approximate them with a number of *light sample points*, and we do visibility tests between a given surface point and each light sample. To keep as many operations in hardware as possible, however, we do not use a hemicube [7] to determine visibility. Instead, to compute the shadows for a single polygon, we render the scene into a scratch buffer, with all polygons except the one being shaded appropriately blackened, using a special projective projection from the point of view of each light sample. These views are registered so that corresponding pixels map to identical points on the polygon. When the resulting hard shadow images are averaged, a soft shadow image results (figure 1). This image is then used directly as a texture on the polygon in order to simulate shadows correctly. The textures so computed are used for real-time display until the scene geometry changes.

In the remainder of the paper, we summarize previous shadow algorithms, we present our method for diffuse scenes in more detail, we discuss generalizations to scenes with specular and general reflectance, we present our implementation and results, and we offer some concluding remarks.

2 Previous Work

2.1 Shadow Algorithms

Woo *et al.* surveyed a number of shadow algorithms [19]. Here we summarize soft shadows methods and methods that run at interactive rates. Shadow algorithms can be divided into three categories: those that compute everything on the fly, those that precompute just visibility, and those that precompute shading.

Computation on the Fly. Simple ray tracing computes everything on the fly. Shadows are computed on a point-by-point basis by tracing rays between the surface point and a point on each light source to check for occluders. Soft shadows can be simulated by tracing rays to a number of points distributed across the light source [8].

The shadow volume approach is another method for computing shadows on the fly. With this method, one constructs imaginary surfaces that bound the shadowed volume of space with respect to each point light source. Determining if a point is in shadow then reduces to point-in-volume testing. Brotman and Badler used an extended z-buffer algorithm with linked lists at each pixel to support soft shadows using this approach [4].

The shadow volume method has also been used in two hardware implementations. Fuchs *et al.* used the pixel processors of the Pixel Planes machine to simulate hard shadows in real-time [10]. Heidmann used the stencil buffer in advanced SGI machines [13]. With Heidmann's algorithm, the scene must be rendered through the stencil created from each light source, so the cost per frame is proportional to the number of light sources times the number of polygons. On 1991 hardware, soft shadows in a fairly simple scene required several seconds with his algorithm. His method appears to be one of the algorithms best suited to interactive use on widely available graphics hardware. We would prefer, however, an algorithm whose cost is sublinear in the number of light sources.

A simple, brute force approach, good for casting shadows of objects onto a plane, is to find the projective transformation that projects objects from a point light onto a plane, and to use it to draw each squashed, blackened object on top of the plane [3], [15, p. 401]. This algorithm effectively multiplies the number of objects in the scene by the number of light sources times the number of *receiver* polygons onto which shadows are being cast, however, so it is typically practical only for very small numbers of light sources and receivers. Another problem with this method is that occluders behind the receiver will cast erroneous shadows, unless extra clipping is done.

Precomputation of Visibility. Instead of computing visibility on the fly, one can precompute visibility from the point of view of each light source.

The z-buffer shadow algorithm uses two (or more) passes of z-buffer rendering, first from the light sources, and then from the eye [18]. The z-buffers from the light views are used in the final

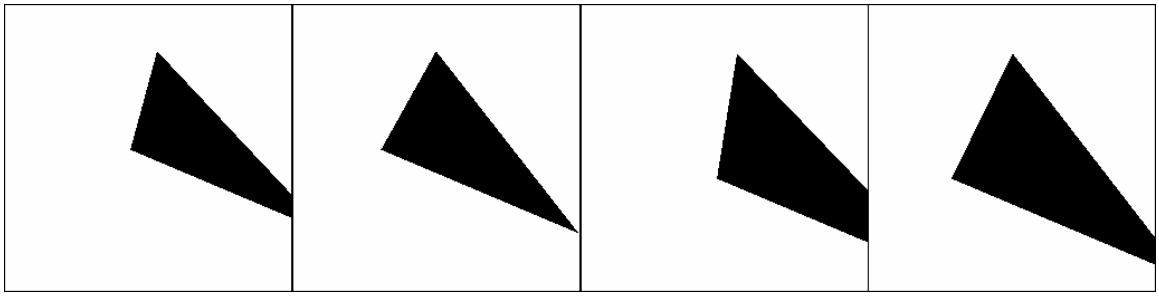


Figure 1: Hard shadow images from 2×2 grid of sample points on light source.

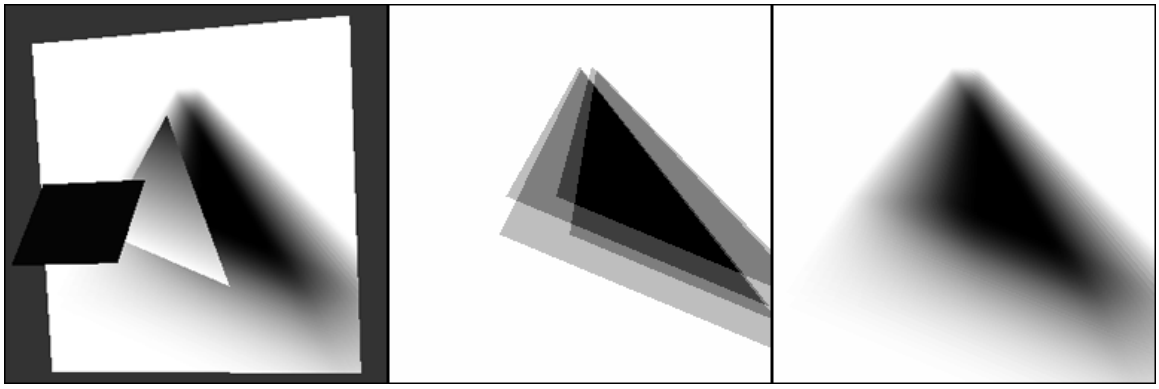


Figure 2: Left: scene with square light source (foreground), triangular occluder (center), and rectangular receiver (background), with shadows on receiver. Center: Approximate soft shadows resulting from 2×2 grid of sample points; the average of the four hard shadow images in Figure 1. Right: Correct soft shadow image (generated with 16×16 sampling). This image is used as the texture on the receiver at left.

pass to determine if a given 3-D point is illuminated with respect to each light source. The transformation of points from one coordinate system to another can be accelerated using texture mapping hardware [17]. This latter method, by Segal *et al.*, achieves real-time rates, and is the other leading method for interactive shadows. Soft shadows can be generated on a graphics workstation by rendering the scene multiple times, using different points on the extended light source, averaging the resulting images using accumulation buffer hardware [11].

A variation of the shadow volume approach is to intersect these volumes with surfaces in the scene to precompute the umbra and penumbra regions on each surface [16]. During the final rendering pass, illumination integrals are evaluated at a sparse sampling of pixels.

Precomputation of Shading. Precomputation can be taken further, computing not just visibility but also shading. This is most relevant to diffuse scenes, since their shading is view-independent. Some of these methods compute visibility continuously, while others compute it discretely.

Several researchers have explored continuous visibility methods for soft shadow computation and radiosity mesh generation. With this approach, surfaces are subdivided into fully lit, penumbra, and umbra regions by splitting along lines or curves where visibility changes. In Chin and Feiner's soft shadow method, polygons are split using BSP trees, and these sub-polygons are then pre-shaded [6]. They achieved rendering times of under a minute for simple scenes. Drettakis and Fiume used more sophisticated computational geometry techniques to precompute their subdivision, and reported rendering times of several seconds [9].

Most radiosity methods discretize each surface into a mesh of elements and then use discrete methods such as ray tracing or hemicubes to compute visibility. The hemicube method computes visibility from a light source point to an entire hemisphere by projecting the scene onto a half-cube [7]. Much of this computation can be done in hardware. Radiosity meshes typically do not resolve shadows well, however. Typical artifacts are Mach bands along the mesh element boundaries and excessively blurry shadows. Most radiosity methods are not fast enough to support interactive changes to the geometry, however. Chen's incremental radiosity method is an exception [5].

Our own method can be categorized next to hemicube radiosity methods, since it also precomputes visibility discretely. Its technique for computing visibility also has parallels to the method of flattening objects to a plane.

2.2 Graphics Hardware

Current graphics hardware, such as the Silicon Graphics Reality Engine [1], can projective-transform, clip, shade, scan convert, and texture tens of thousands of polygons in real-time (in 1/30 sec.). We would like to exploit the speed of this hardware to simulate soft shadows.

Typically, such hardware supports arbitrary 4×4 homogeneous transformations of planar polygons, clipping to any truncated pyramidal frustum (right or oblique), and scan conversion with z-buffering or overwriting. On SGI machines, Phong shading (once per pixel) is not possible, but faceted shading (once per polygon) and Gouraud shading (once per vertex) are supported. Phong shading

can be simulated by splitting polygons into small pieces on input. A common, general form for hardware-supported illumination is diffuse reflection from multiple point spotlight sources, with a texture mapped reflectance function and attenuation:

$$I_c(x,y) = T_c(u,v) \sum_l \frac{\cos\theta_l \cos\theta_l'^e \, L_{lc}}{\alpha + \beta r_l + \gamma r_l^2}$$

where c is color channel index (= r, g, or b), $I_c(x,y)$ is the pixel value at screen space (x,y), $T_c(u,v)$ is a texture parameterized by texture coordinates (u,v), which are a projective transform of (x,y), θ_l is the polar angle for the ray to light source l, θ_l' is the angle away from the directional axis of the light source, e is the spotlight exponent, L_{lc} is the radiance of light l, r_l is distance to light source l, and α, β, and γ are constants controlling attenuation. Texture mapping, lights, and attenuation can be turned on and off independently on a per-polygon basis. Most systems also support Phong illumination, which has an additional specular term that we have not shown. The most advanced, expensive machines support all of these functions in hardware, while the cheaper machines do some of these calculations in software. Since the graphics subroutine interface, such as OpenGL [15], is typically identical on any machine, these differences are transparent to the user, except for the dramatic differences in running speed. So when we speak of a computation being done "in hardware", that is true only on high end machines.

The accumulation buffer [11], another feature of some graphics workstations, is hardware that allows a linear combination of images to be easily computed. It is capable of computing expressions of the general form:

$$A_c(x,y) = \sum_i \alpha_i I_{ic}(x,y)$$

where I_{ic} is a channel of image i, and A_c is a channel of the accumulator array.

3 Diffuse Scenes

Our shadow generation method for diffuse scenes takes advantage of these hardware capabilities.

Direct illumination in a scene of opaque surfaces that emit or reflect light diffusely is given by the following formula:

$$L_c(\mathbf{x}) = \rho_c(\mathbf{x}) \left(L_{\text{ac}} + \int_{\text{lights}} \frac{\cos_+\theta \cos_+\theta'}{\pi r^2} \, v(\mathbf{x},\mathbf{x}') \, L_c(\mathbf{x}') \, d\mathbf{x}' \right),$$

where, as shown in Figure 3,

- $\mathbf{x} = (x,y,z)$ is a 3-D point on a reflective surface, and $\mathbf{x}'$ is a point on a light source,
- θ is polar angle (angle from normal) at $\mathbf{x}$, θ' is the angle at $\mathbf{x}'$,
- r is the distance between $\mathbf{x}$ and $\mathbf{x}'$,
- θ, θ', and r are functions of $\mathbf{x}$ and $\mathbf{x}'$,
- $L_c(\mathbf{x})$ is outgoing radiance at point $\mathbf{x}$ for color channel c, due to either emission or reflection, L_{ac} is ambient radiance,
- $\rho_c(\mathbf{x})$ is reflectance,
- $v(\mathbf{x},\mathbf{x}')$ is a Boolean visibility function that equals 1 if point $\mathbf{x}$ is visible from point $\mathbf{x}'$, else 0,
- $\cos_+\theta = \max(\cos\theta, 0)$, for backface testing, and
- the integral is over all points on all light sources, with respect to $d\mathbf{x}'$, which is an infinitesimal area on a light source.

The inputs to the problem are the geometry, the reflectance $\rho_c(\mathbf{x})$, and emitted radiance $L_c(\mathbf{x}')$ on all light sources, the ambient radiance L_{ac}, and the output is the reflected radiance function $L_c(\mathbf{x})$.

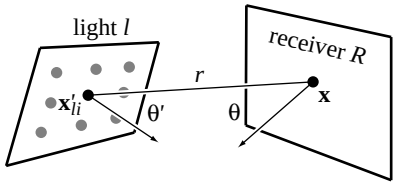


Figure 3: Geometry for direct illumination. The radiance at point $\mathbf{x}$ on the receiver is being calculated by summing the contributions from a set of point light sources at $\mathbf{x}'_{li}$ on light l.

3.1 Approximating Extended Light Sources

Although such integrals can be solved in closed form for planar surfaces with no occlusion ($v \equiv 1$), the complexity of the visibility function makes these integrals intractable in the general case. We can compute approximations to the integral, however, by replacing each extended light source l by a set of n_l point light sources:

$$L_c(\mathbf{x}') \approx \sum_l \sum_{i=1}^{n_l} a_{li} \, L_c(\mathbf{x}') \, \delta(\mathbf{x}' - \mathbf{x}'_{li}),$$

where $\delta(\mathbf{x})$ is a 3-D Dirac delta function, $\mathbf{x}'_{li}$ is sample point i on light source l, and a_{li} is the area associated with this sample point. Typically, each sample on a light source has equal area: $a_{li} = a_l / n_l$, where a_l is the area of light source l.

With this approximation, the radiance of a reflective surface point can be computed by summing the contributions over all sample points on all light sources:

$$\begin{aligned} L_c(\mathbf{x}) &= \rho_c(\mathbf{x}) \, L_{\text{ac}} \\ &\quad + \rho_c(\mathbf{x}) \sum_l \sum_{i=1}^{n_l} a_{li} \frac{\cos_+\theta_{li} \cos_+\theta'_{li}}{\pi r_{li}^2} \, v(\mathbf{x},\mathbf{x}'_{li}) \, L_c(\mathbf{x}'_{li}). \end{aligned} \tag{1}$$

The formulas above can be generalized to linear and point light sources, as well as area light sources.

The most difficult and expensive part of the above calculation is evaluation of the visibility function v, since it requires global knowledge of the scene, whereas the remaining factors require only local knowledge. But computing v is necessary in order to simulate shadows. The above formula could be evaluated using ray tracing, but the resulting algorithm would be slow due to the large number of light source samples.

3.2 Soft Shadows in Hardware

Equation (1) can be rewritten in a form suitable to hardware computation:

$$\begin{aligned} L_c(\mathbf{x}) &= \rho_c(\mathbf{x}) \, L_{\text{ac}} \\ &\quad + \sum_l \sum_{i=1}^{n_l} \left(a_{li} \, \rho_c(\mathbf{x}) \right) \left(\frac{\cos_+\theta_{li} \cos_+\theta'_{li} \, L_c(\mathbf{x}'_{li})}{\pi r_{li}^2} \right) v(\mathbf{x},\mathbf{x}'_{li}). \end{aligned} \tag{2}$$

Each term in the inner summation can be regarded as a hard shadow image resulting from a point light source at $\mathbf{x}'_{li}$, where $\mathbf{x}$ is a function of screen (x,y).

That summand consists of the product of three factors. The first one, which is an area times the reflectance of the receiving polygon, can be calculated in software. The second factor is the cosine of the angle on the receiver, times the cosine of the angle on the light

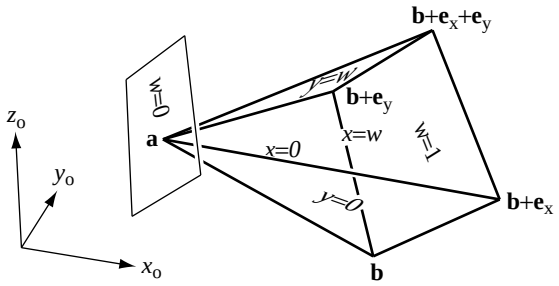


Figure 4: Pyramid with parallelogram base. Faces of pyramid are marked with their plane equations.

source, times the radiance of the light source, divided by r^2. This can be computed in hardware by rendering the receiver polygon with a single spotlight at $\mathbf{x}'_{li}$ turned on, using a spotlight exponent of $e = 1$ and quadratic attenuation. On machines that do not support Phong shading, we will have to finely subdivide the polygon. The third factor is visibility between a point on a light source and each point on the receiver. Visibility can be computed by projecting all polygons between light source point $\mathbf{x}'_{li}$ and the receiver onto the receiver.

We want to simulate soft shadows as quickly as possible. To take full advantage of the hardware, we can precompute the shading for each polygon using the formula above, and then display views of the scene from moving viewpoints using real-time texture mapping and z-buffering.

To compute soft shadow textures, we need to generate a number of hard shadow images and then average them. If these hard shadow images are not registered (they would not be, using hemi-cubes), then it would be necessary to resample them so that corresponding pixels in each hard shadow image map to the same surface point in 3-D. This would be very slow. A faster alternative is to choose the transformation for each projection so that the hard shadow images are perfectly registered with each other.

For planar receiver surfaces, this is easily accomplished by exploiting the capabilities of projective transformations. If we fit a parallelogram around the receiver surface of interest, and then construct a pyramid with this as its base and the light point as its apex, there is a 4×4 homogeneous transformation that will map such a pyramid into an axis-aligned box, as described shortly.

The hard shadow image due to sample point i on light l is created by loading this special transformation matrix and rendering the receiver polygon. The polygon is illuminated by the ambient light plus a single point light source at $\mathbf{x}'_{li}$, using Phong shading or a good approximation to it. The visibility function is then computed by rendering the remainder of the scene with all surfaces shaded as if they were the receiver illuminated by ambient light: $(r, g, b) = (\rho_r L_{ar}, \rho_g L_{ag}, \rho_b L_{ab})$. This is most quickly done with z-buffering off, and clipping to a pyramid with the receiver polygon as its base. Drawing each polygon with an unsorted painter's algorithm suffices here because all polygons are the same color, and after clipping, the only polygon fragments remaining will lie between the light source and the receiver, so they all cast shadows on the receiver. To compute the weighted average of the hard shadow images so created, we use the accumulation buffer.

3.3 Projective Transformation of a Pyramid to a Box

We want a projective (perspective) transformation that maps a pyramid with parallelogram base into a rectangular parallelepiped. The pyramid lies in object space, with coordinates (x_o, y_o, z_o). It has apex $\mathbf{a}$ and its parallelogram base has one vertex at $\mathbf{b}$ and edge vectors $\mathbf{e}_x$ and $\mathbf{e}_y$ (bold lower case denotes a 3-D point or vector). The parallelepiped lies in what we will call unit screen space, with coordinates (x_u, y_u, z_u). Viewed from the apex, the left and right sides of the pyramid map to the parallel planes $x_u = 0$ and $x_u = 1$, the bottom and top map to $y_u = 0$ and $y_u = 1$, and the base plane and a plane parallel to it through the apex map to $z_u = 1$ and $z_u = \infty$, respectively. See figure 4.

A 4×4 homogeneous matrix effecting this transformation can be derived from these conditions. It will have the form:

$$\mathbf{M} = \begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ 0 & 0 & 0 & 1 \\ m_{30} & m_{31} & m_{32} & m_{33} \end{pmatrix},$$

and the homogeneous transformation and homogeneous division to transform object space to unit screen space are:

$$\begin{pmatrix} x \\ y \\ 1 \\ w \end{pmatrix} = \mathbf{M} \begin{pmatrix} x_o \\ y_o \\ z_o \\ 1 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} x_u \\ y_u \\ z_u \end{pmatrix} = \begin{pmatrix} x/w \\ y/w \\ 1/w \end{pmatrix}.$$

The third row of matrix $\mathbf{M}$ takes this simple form because a constant z_u value is desired on the base plane. The homogeneous screen coordinates x, y, and w are each affine functions of x_o, y_o, and z_o (that is, linear plus translation). The constraints above specify the value of each of the three coordinates at four points in space – just enough to uniquely determine the twelve unknowns in $\mathbf{M}$.

The w coordinate, for example, has value 1 at the points $\mathbf{b}$, $\mathbf{b}+\mathbf{e}_x$, and $\mathbf{b}+\mathbf{e}_y$, and value 0 at $\mathbf{a}$. Therefore, the vector $\mathbf{n}_w = \mathbf{e}_y \times \mathbf{e}_x$ is normal to any plane of constant w, thus fixing the first three elements of the last row of the matrix within a scale factor: $(m_{30}, m_{31}, m_{32})^T = \alpha_w \mathbf{n}_w$. Setting w to 0 at $\mathbf{a}$ and 1 at $\mathbf{b}$ constrains $m_{33} = -\alpha_w \mathbf{n}_w \cdot \mathbf{a}$ and $\alpha_w = 1/\mathbf{n}_w \cdot \mathbf{e}_w$, where $\mathbf{e}_w = \mathbf{b} - \mathbf{a}$. The first two rows of $\mathbf{M}$ can be derived similarly (see figure 4). The result is:

$$\mathbf{M} = \begin{pmatrix} \alpha_x n_{xx} & \alpha_x n_{xy} & \alpha_x n_{xz} & -\alpha_x \mathbf{n}_x \cdot \mathbf{b} \\ \alpha_y n_{yx} & \alpha_y n_{yy} & \alpha_y n_{yz} & -\alpha_y \mathbf{n}_y \cdot \mathbf{b} \\ 0 & 0 & 0 & 1 \\ \alpha_w n_{wx} & \alpha_w n_{wy} & \alpha_w n_{wz} & -\alpha_w \mathbf{n}_w \cdot \mathbf{a} \end{pmatrix},$$

where

$$\begin{array}{lll} \mathbf{n}_x = \mathbf{e}_w \times \mathbf{e}_y & & \alpha_x = 1/\mathbf{n}_x \cdot \mathbf{e}_x \\ \mathbf{n}_y = \mathbf{e}_x \times \mathbf{e}_w & \text{and} & \alpha_y = 1/\mathbf{n}_y \cdot \mathbf{e}_y \\ \mathbf{n}_w = \mathbf{e}_y \times \mathbf{e}_x & & \alpha_w = 1/\mathbf{n}_w \cdot \mathbf{e}_w \end{array}.$$

Blinn [3] uses a related projective transformation for the generation of shadows on a plane, but his is a projection (it collapses 3-D to 2-D), while ours is 3-D to 3-D. We use the third dimension for clipping.

3.4 Using the Transformation

To use this transformation in our shadow algorithm, we first fit a parallelogram around the receiver polygon. If the receiver is a rectangle or other parallelogram, the fit is exact; if the receiver is a triangle, then we fit the triangle into the lower left triangle of the parallelogram; and for more general polygons with four or more sides, a simple 2-D bounding box in the plane of the polygon can be used. It is possible to go further with projective transformations, mapping arbitrary planar quadrilaterals into squares (using the homogeneous texture transformation matrix of OpenGL, for example). We assume for simplicity, however, that the transformation between texture space (the screen space in these light source projections) and object space is affine, and so we restrict ourselves to parallelograms.

3.5 Soft Shadow Algorithm for Diffuse Scenes

To precompute soft shadow radiance textures:

```
turn off z-buffering
for each receiver polygon R
  choose resolution for receiver's texture (s_x × s_y pixels)
  clear accumulator image of s_x × s_y pixels to black
  create temporary image of s_x × s_y pixels
  for each light source l
    first backface test: if l is entirely behind R
      or R is entirely behind l, then skip to next l
    for each sample point i on light source l
      second backface test: if x'_li is behind R then skip to next i
      compute transformation matrix M, where a = x'_li,
        and the base parallelogram fits tightly around R
      set current transformation matrix to scale(s_x, s_y, 1)·M
      set clipping planes to z_u,near = 1 − ε and z_u,far = big
      draw R with illumination from x'_li only, as described in
        equation (2), into temp image
      for each other object in scene
        draw object with ambient color into temp image
      add temp image into accumulator image with weight a_l / n_l
  save accumulator image as texture for polygon R
```

A hard shadow image is computed in each iteration of the i loop. These are averaged together to compute a soft shadow image, which is used as a radiance texture. Note that objects casting shadows need not be polygonal; any object that can be quickly scan converted will work well.

To display a static scene from moving viewpoints, simply:

```
turn on z-buffering
for each object in scene
  if object receives shadows, draw it textured but without illumination
  else draw object with illumination
```

3.6 Backface Testing

The cases where $\cos_+\theta\ \cos_+\theta' = 0$ can be optimized using backface testing.

To test if polygon p is behind polygon q, compute the signed distances from the plane of polygon q to each of the vertices of p (signed positive on the front of q and negative on the back). If they are all positive, then p is entirely in front of q, if they are all nonpositive, p is entirely in back, otherwise, part of p is in front of q and part is in back.

To test if the apex $\mathbf{a}$ of the pyramid is behind the receiver R that defines the base plane, simply test if $\mathbf{n}_w \cdot \mathbf{e}_w \leq 0$.

The above checks will ensure that $\cos\theta > 0$ at every point on the receiver, but there is still the possibility that $\cos\theta' \leq 0$ on portions of the receiver (i.e. that the receiver is only partially illuminated by the light source). This final case should be handled at the polygon level or pixel level when shading the receiver in the algorithm above. Phong shading, or a good approximation to it, is needed here.

3.7 Sampling Extended Light Sources

The set of samples used on each light source greatly influences the speed and quality of the results. Too few samples, or a poorly chosen sample distribution, result in penumbras that appear stepped, not continuous. If too many samples are used, however, the simulation runs too slowly.

If a uniform grid of sample points is used, the stepping is much more pronounced in some cases. For example, if a uniform grid of $m \times m$ samples is used on a parallelogram light source, an occluder edge coplanar with one of the light source edges will cause m big steps, while an occluder edge in general position will cause m^2 small steps.

Stochastic sampling [8] with the same number of samples yields smoother penumbra than a uniform grid, because the steps no longer coincide. We use a jittered uniform grid because it gives good results and is very easy to compute.

Using a fixed number of samples on each light source is inefficient. Fine sampling of a light source is most important when the light source subtends a large solid angle from the point of view of the receiver, since that is when the penumbra is widest and stepping artifacts would be most visible. A good approach is to choose the light source sample resolution such that the solid angle subtended by the light source area associated with each sample is below a user-specified threshold.

The algorithm can easily handle diffuse (non-directional) light sources whose outgoing radiance varies with position, such as stained glass windows. For such light sources, importance sampling might be preferable: concentration of samples in the regions of the light source with highest radiance.

3.8 Texture Resolution

The resolution of the shadow texture should be roughly equal to the resolution at which it will be viewed (one texture pixel mapping to one screen pixel); lower resolution results in visible artifacts such as blocky shadows, and higher resolution is wasteful of time and memory. In the absence of information about probable views, a reasonable technique is to set the number of pixels on a polygon's texture, in each dimension, proportional to its size in world space using a "desired pixel size" parameter. With this scheme, the required texture memory, in pixels, will be the total world space surface area of all polygons in the scene divided by the square of the desired pixel size.

Texture memory for triangles can be further optimized by packing the textures for two triangles into one rectangular texture block.

If there are too many polygons in the scene, or the desired pixel size is too small, the texture memory could be exceeded, causing paging of texture memory and slow performance.

Radiance textures can be antialiased by supersampling: generating the hard and initial soft shadow images at several times the desired resolution, and then filtering and downsampling the images before creating textures. Textured surfaces should be rendered with good texture filtering.

Some polygons will contain penumbral regions with respect to a light source, and will require high texture resolution, but others will be either totally shadowed (umbral) or totally illuminated by each light source, and will have very smooth radiance functions. Sometimes these functions will be so smooth that they can be adequately approximated by a single Gouraud shaded polygon. This optimization saves significant texture memory and speeds display.

This idea can be carried further, replacing the textured planar polygon with a mesh of coplanar Gouraud shaded triangles. For complex shadow patterns and radiance functions, however, textures may render faster than the corresponding Gouraud approximation, depending on the relative speed of texture mapping and Gouraud-shaded triangle drawing, and the number of triangles required to achieve a good approximation.

3.9 Complexity

We now analyze the expected complexity of our algorithm (worst case costs are not likely to be observed in practice, so we do not discuss them here). Although more sophisticated schemes are possible, we will assume for the purposes of analysis that the same set

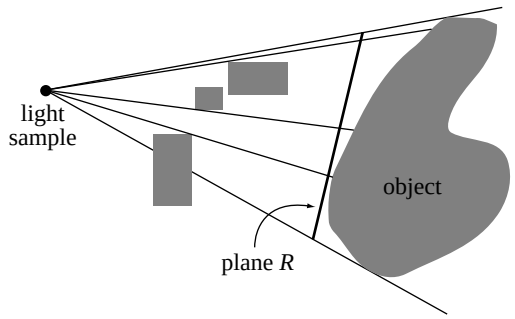


Figure 5: Shadows are computed on plane R and projected onto the receiving object at right.

of light samples are used for shadowing all polygons. Suppose we have a scene with s surfaces (polygons), a total of $n = \sum_l n_l$ light source samples, a total of t radiance texture pixels, and the output images are rendered with p pixels. We assume the depth complexity of the scene (the average number of surfaces intersecting a ray) is bounded, and that t and p are roughly linearly related. The average number of texture pixels per polygon is t/s.

With our technique, preprocessing renders the scene ns times. A painter's algorithm rendering of s polygons into an image of t/s pixels takes $O(s+t/s)$ time for scenes of bounded depth complexity. The total preprocessing time is thus $O(ns^2+nt)$, and the required texture memory is $O(t)$. Display requires only z-buffered texture mapping of s polygons to an image of p pixels, for a time cost of $O(s+p)$. The memory for the z-buffer and output image is $O(p) = O(t)$.

Our display algorithm is very fast for complex scenes. Its cost is independent of the number of light source samples used, and also independent of the number of texture pixels (assuming no texture paging).

For scenes of low or moderate complexity, our preprocessing algorithm is fast because all of its pixel operations can be done in hardware. For very complex scenes, our preprocessing algorithm becomes impractical because it is quadratic in s, however. In such cases, performance can be improved by calculating shadows only on a small number of surfaces in the scene (e.g. floor, walls, and other large, important surfaces), thereby reducing the cost to $O(nss_t+nt)$, where s_t is the number of textured polygons.

In an interactive setting, a progressive refinement of images can be used, in which hard shadows on a small number of polygons (precomputation with $n = 1$, s_t small) are rendered while the user is moving objects with the mouse, a full solution (precomputation with n large, s_t large) is computed when they complete a movement, and then top speed rendering (display with texture mapping) is used as the viewer moves through the scene.

More fundamentally, the quadratic cost can be reduced using more intelligent data structures. Because the angle of view of most of the shadow projection pyramids is narrow, only a small fraction of the polygons in a scene shadow a given polygon, on average. Using spatial data structures, entire objects can be culled with a few quick tests [2], obviating transformation and clipping of most of the scene, speeding the rendering of each hard shadow image from $O(s+t/s)$ to $O(s^\alpha+t/s)$, where $\alpha \approx .3$ or so.

An alternative optimization, which would make the algorithm more practical for the generation of shadows on complex curved or many-faceted objects, is to approximate a receiving object with a plane, compute shadows on this plane, and then project the shadows onto the object (figure 5). This has the advantage of replacing many renderings with a single rendering, but its disadvantage is that self-shadowing of concave objects is not simulated.

3.10 Comparison to Other Algorithms

We can compare the complexity of our algorithm to other algorithms capable of simulating soft shadows at near-interactive rates. The main alternatives are the stencil buffer technique by Heidmann, the z-buffer method by Segal *et al.*, and hardware hemicube-based radiosity algorithms.

The stencil buffer technique renders the scene once for each light source, so its cost per frame is $O(ns+np)$, making it difficult to support soft shadows in real-time. With the z-buffer shadow algorithm, the preprocessing time is acceptable, but the memory cost and display time cost are $O(np)$. This makes the algorithm awkward for many point light sources or extended light sources with many samples (large n). When soft shadows are desired, our approach appears to yield faster walkthroughs than either of these two methods, because our display process is so fast.

Among current radiosity algorithms, progressive radiosity using hardware hemicubes is probably the fastest method for complex scenes. With progressive radiosity, very high resolution hemicubes and many elements are needed to get good shadows, however. While progressive radiosity may be a better approach for shadow generation in very complex scenes (very large s), it appears slower than our technique for scenes of moderate complexity because every pixel-level operation in our algorithm can be done in hardware, but this is not the case with hemicubes, since the process of summing differential form factors while reading out of the hemicube must be done in software [7].

4 Scenes with General Reflectance

Shadows on specular surfaces, or surfaces with more general reflectance, can be simulated with a generalization of the diffuse algorithm, but not without added time and memory costs.

Shadows from a single point light source are easily simulated by placing just the visibility function $v(\mathbf{x}, \mathbf{x}')$ in texture memory, creating a Boolean *shadow texture*, and computing the remaining local illumination factors at vertices only. This method costs $O(ss_t+t)$ for precomputation, and $O(s+p)$ for display.

Shadows from multiple point light sources can also be simulated. After precomputing a shadow texture for each polygon when illuminated with each light source, the total illumination due to n light sources can be calculated by rendering the scene n times with each of these sets of shadow textures, compositing the final image using blending or with the accumulation buffer. The cost of this method is nt one-bit texture pixels and $O(ns+np)$ display time.

Generalizing this method to extended light sources in the case of general reflectance is more difficult, as the computation involves the integration of light from polygonal light sources weighted by the bidirectional reflectance distribution functions (BRDFs). Specular BRDF's are spiky, so careful integration is required or the highlights will betray the point sampling of the light sources. We believe, however, that with careful light sampling and numerical integration of the BRDF's, soft shadows on surfaces with general reflectance could be displayed with $O(nt)$ memory and $O(ns+np)$ time.

5 Implementation

We implemented our diffuse algorithm using the OpenGL subroutine library, running with the IRIX 5.3 operating system on an SGI Crimson with 100 MHz MIPS R4000 processor and Reality Engine graphics. This machine has hardware for texture mapping and an accumulation buffer with 24 bits per channel.

The implementation is fairly simple, since OpenGL supports loading of arbitrary 4×4 matrices, and we intentionally cast our

shading formulas in a form that maps cleanly into OpenGL's model. The source code is about 2,000 lines of C++. Our implementation renders at about 900×900 resolution, and uses 24-bit textures at sizes of $2^{k_x} \times 2^{k_y}$ pixels, for $2 \leq k_x, k_y \leq 8$. Phong shading is simulated by subdividing each receiver polygon into a grid of 8×8-pixel parallelograms during preprocessing.

Our software allows interactive movement of objects and the camera. When the scene geometry is changed, textures are recomputed. On a scene with $s = 749$ polygons, $s_t = 3$ of them textured, with two area light sources sampled with $n = 8$ points total, generating textures with about $t = 200,000$ pixels total, and a final picture of about $p = 810,000$ pixels, preprocessing has a redisplay rate of 2 Hz. For simple scenes, the slowest part of preprocessing is the transfer of radiance textures from system memory to texture memory.

When only the view is changed, we simply redisplay the scene with texture mapping. The use of OpenGL display lists helps us achieve 30 Hz rates in most cases. When we allocate more radiance texture memory than the hardware can hold, however, paging slows redisplay.

Since we know the size and perceptual importance of each object at modeling time, we have found it convenient to have each receiver object control the number of light source samples that are used to illuminate it. The floor and walls, for example, might specify many light source samples, while table and chairs might specify a single light source sample. To facilitate further testing of shadow sampling, a slider that acts as a multiplier on the requested number of samples per light source is provided. More automatic and intelligent light sampling schemes are certainly possible.

6 Results

The color figures illustrate high quality results achievable in a few seconds with fine light source sampling. Figure 6 shows a scene with 6,142 polygons, 3 of them shadowed, which was computed in 5.5 seconds using $n = 32$ light samples total on two light sources. Figure 7 illustrates the calculation of shadows on more complex objects, with a total of $s_t = 25$ shadowed polygons. For this image, 7×7 light sampling was used when shadowing the walls and floor, while 3×3 sampling was used to compute shadows on the table top, and 2×2 sampling was used for the table legs. The textures for the table polygons are smaller than those for the walls and floor, in proportion to their world space size. This image was calculated in 13 seconds.

7 Conclusions

We have described a simple algorithm for generating soft shadows at interactive rates by exploiting graphics workstation hardware. Previous shadow generation methods have not supported both the computation and display of soft shadows at these speeds.

To achieve real time rates with our method, one probably needs hardware support for transformation, clipping, scan conversion, texture mapping, and accumulation buffer operations. In coming years, such hardware will only become more affordable, however. Software implementations will also work, of course, but at reduced speeds.

For most scenes, realistic images can be generated by computing soft shadows only for a small set of polygons. This will run quite fast. If it is necessary to compute shadows for every polygon, our preprocessing method has quadratic growth with respect to scene complexity s, but we believe this can be reduced to about $O(s^{1.3})$, using spatial data structures to cull off-screen objects.

Once preprocessing is done, the display cost is independent of the number and size of light sources. This cost is little more than the display cost without shadows.

The method also has potential as a form factor calculation technique for progressive radiosity.

8 Acknowledgments & Notes

We thank Silicon Graphics for the gift of a Reality Engine, which made this work possible. Jeremiah Blatz and Michael Garland provided modeling assistance. This paper grew out of a project by Herf in a graduate course on Rendering taught by Heckbert, Fall 1995.

References

[1] Kurt Akeley. RealityEngine graphics. In *SIGGRAPH '93 Proc.*, pages 109–116, Aug. 1993.

[2] James Arvo and David Kirk. A survey of ray tracing acceleration techniques. In Andrew S. Glassner, editor, *An introduction to ray tracing*, pages 201–262. Academic Press, 1989.

[3] James F. Blinn. Me and my (fake) shadow. *IEEE Computer Graphics and Applications*, 8(1):82–86, Jan. 1988.

[4] Lynne Shapiro Brotman and Norman I. Badler. Generating soft shadows with a depth buffer algorithm. *IEEE Computer Graphics and Applications*, 4(10):5–24, Oct. 1984.

[5] Shenchang Eric Chen. Incremental radiosity: An extension of progressive radiosity to an interactive image synthesis system. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):135–144, August 1990.

[6] Norman Chin and Steven Feiner. Fast object-precision shadow generation for area light sources using BSP trees. In *1992 Symp. on Interactive 3D Graphics*, pages 21–30. ACM SIGGRAPH, Mar. 1992.

[7] Michael F. Cohen and Donald P. Greenberg. The hemi-cube: A radiosity solution for complex environments. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):31–40, July 1985.

[8] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Trans. on Graphics*, 5(1):51–72, Jan. 1986.

[9] George Drettakis and Eugene Fiume. A fast shadow algorithm for area light sources using backprojection. In *SIGGRAPH '94 Proc.*, pages 223–230, 1994. http://safran.imag.fr/Membres/George.Drettakis/pub.html.

[10] Henry Fuchs, Jack Goldfeather, Jeff P. Hultquist, Susan Spach, John D. Austin, Frederick P. Brooks, Jr., John G. Eyles, and John Poulton. Fast spheres, shadows, textures, transparencies, and image enhancements in Pixel-Planes. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):111–120, July 1985.

[11] Paul Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):309–318, Aug. 1990.

[12] Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):145–154, Aug. 1990.

[13] Tim Heidmann. Real shadows, real time. *Iris Universe*, 18:28–31, 1991. Silicon Graphics, Inc.

[14] Karol Myszkowski and Tosiyasu L. Kunii. Texture mapping as an alternative for meshing during walkthrough animation. In *Fifth Eurographics Workshop on Rendering*, pages 375–388, June 1994.

[15] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley, Reading MA, 1993.

[16] Tomoyuki Nishita and Eihachiro Nakamae. Half-tone representation of 3-D objects illuminated by area sources or polyhedron sources. In *COMPSAC '83, Proc. IEEE 7th Intl. Comp. Soft. and Applications Conf.*, pages 237–242, Nov. 1983.

[17] 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.

[18] Lance Williams. Casting curved shadows on curved surfaces. *Computer Graphics (SIGGRAPH '78 Proceedings)*, 12(3):270–274, Aug. 1978.

[19] Andrew Woo, Pierre Poulin, and Alain Fournier. A survey of shadow algorithms. *IEEE Computer Graphics and Applications*, 10(6):13–32, Nov. 1990.

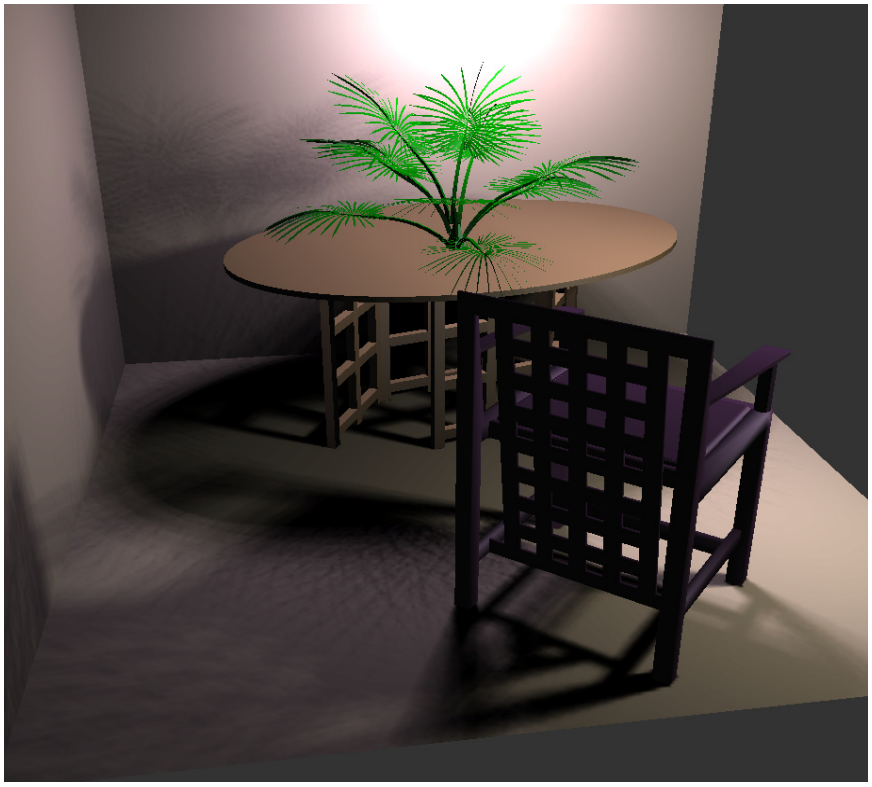


Figure 6: Shadows on walls and floor, computed in 5.5 seconds.

Figure 7: Shadows on walls, floor, and table, computed in 13 seconds.

Volume 15, (1997) number 4 pp. 249–261

Interactive Rendering of CSG Models

T. F. Wiegand†

The Martin Centre for Architectural and Urban Studies
The University of Cambridge, Cambridge, UK

Abstract

We describe a CSG rendering algorithm that requires no evaluation of the CSG tree beyond normalization and pruning. It renders directly from the normalized CSG tree and primitives described (to the graphics system) by their facetted boundaries. It behaves correctly in the presence of user defined, "near" and "far" clipping planes. It has been implemented on standard graphics workstations using Iris GL ? and OpenGL ? graphics libraries. Modestly sized models can be evaluated and rendered at interactive (less than a second per frame) speeds. We have combined the algorithm with an existing B-rep based modeller to provide interactive rendering of incremental updates to large models.

1. Introduction

Constructive Solid Geometry (CSG) within an interactive modelling environment provides a simple and intuitive approach to solid modelling. In conventional modelling systems primitives are first positioned, a boolean operation is performed and the results then rendered. Often the correct position cannot be gauged easily from display of the primitives alone. A sequence of trial and error may be initiated or perhaps a break from the normal modelling process to calculate the correct position numerically. Conceptual modelling is inhibited — usually a design is fully fledged before modelling commences. Interactive rendering offers the promise of a modelling system where designers can easily explore possibilities within the CSG paradigm. For instance, a designer could drag a hole defined by a complex solid through a workpiece, observing the new forms that emerge.

Interactive rendering of CSG models has previously been implemented with special purpose hardware ?, ?, ?. We believe that such systems should be based on an existing, commonly available graphics library. Use of an existing graphics library simplifies development, protects investment in proprietary graphics hardware, and leverages off future improvements in the hardware supported by the library. Conversion of the CSG tree for a model into a boundary representation (B-rep) meets this goal but is typically too slow for interactive modification.

The surfaces in the B-rep of a model are a subset of the surfaces of the primitives in the CSG tree for the model. Conversion to a B-rep is then the classification of the surfaces of each primitive into portions that are "inside", "outside", or "on" the surface of the fully evaluated model. Display of the model only requires classification of the points on the surfaces which project to each pixel. Point classification is much simpler than surface classification. Geometrically, point classification requires intersection of the primitives with rays through each pixel, while surface classification requires intersection of the primitive surfaces with each other.

Thibault and Naylor ? describe a surface classification based approach. They build BSP trees for each primitive and perform the classification by merging the trees together. The resulting tree is equivalent to a BSP tree built from the B-rep of the model. The complete evaluation process is too slow for interactive rendering. They describe an incremental version of their algorithm which provides interactive rendering speeds within a modelling environment.

There are variations of most rendering algorithms which use point classification. These include ray trac-

† Supported by Informatix, Inc. Tokyo.

© The Eurographics Association 1997. Published by Blackwell Publishers, 108 Cowley Road, Oxford OX4 1JF, UK and 238 Main Street, Cambridge, MA 02142, USA.

ing [?], scan line methods [?], and depth-buffer methods [?, ?, ?]. Much attention has been focused on optimising point classification for this purpose [?]. These algorithms all add point classification within the lowest levels of the standard algorithms. We require an algorithm which can be implemented using an existing graphics library.

Goldfeather, Molnar, Turk and Fuchs [?] describe an algorithm that first normalizes a CSG tree before rendering the normalized form. It operates in a SIMD pixel parallel way on an augmented frame buffer (Pixel-planes 4) which has two depth (Z) buffers, two color buffers and flag bits per pixel. We have developed a new version of this algorithm capable of being implemented using an existing graphics library on a conventional graphics workstation. Our algorithm requires a single depth buffer, single color buffer, stencil (flag bits) buffer and the ability to save and restore the contents of the depth buffer.

In section **??** we review the algorithm described by Goldfeather et. al. [?]. We have restructured the presentation of the ideas to make them more amenable to implementation on a conventional graphics workstation. Our implementation is described in section **??**. In section **??** we describe the integration of user defined, "near" and "far" clipping planes into the algorithm. In section **??** we describe use of the algorithm within an interactive modelling system. The system maintains fully evaluated B-rep versions of models and uses the rendering algorithm for interactive changes to the models. Section **??** presents performance statistics for our current implementation using the Silicon Graphics GL library [?].

2. Rendering a CSG tree using pixel parallel operations

We would advise interested readers to refer to Goldfeather et. al. [?] for a fuller description of the algorithm which we summarize in this section.

A CSG tree is either a primitive or a boolean combination of sub-trees with intersection($\cap$), subtraction($-$) or union($\cup$) operators. A CSG tree is in normal (sum of products) form when all intersection or subtraction operators have a left subtree which contains no union operators and a right subtree that is simply a primitive. For example $(((A \cap B) - C) \cup (D \cap (E - (F \cap G)))) \cup H$, where A–H represent primitives, is in normal form. We shall assume left association of operators so the previous expression can be written as $(A \cap B - C) \cup (D \cap E - F \cap G) \cup H$. This expression has three products. The primitives A, B, D, E, G, H are *uncomplemented*, C and F are *complemented*.

The normalization process recursively applies a set of production rules to a CSG tree which use the associative and distributive properties of boolean operations. Determining an appropriate rule and applying it uses only local information (type of current node and child node types). The production rules and algorithm used are :

1. $X - (Y \cup Z) \rightarrow (X - Y) - Z$
2. $X \cap (Y \cup Z) \rightarrow (X \cap Y) \cup (X \cap Z)$
3. $X - (Y \cap Z) \rightarrow (X - Y) \cup (X - Z)$
4. $X \cap (Y \cap Z) \rightarrow (X \cap Y) \cap Z$
5. $X - (Y - Z) \rightarrow (X - Y) \cup (X \cap Z)$
6. $X \cap (Y - Z) \rightarrow (X \cap Y) - Z$
7. $(X - Y) \cap Z \rightarrow (X \cap Z) - Y$
8. $(X \cup Y) - Z \rightarrow (X - Z) \cup (Y - Z)$
9. $(X \cup Y) \cap Z \rightarrow (X \cap Z) \cup (Y \cap Z)$

```
proc normalize(T : tree)
{
    if T is a primitive {
      return
    }
    repeat {
        while T matches a rule from 1–9 {
            apply first matching rule
        }
        normalize(T.left)
    } until (T.op is a union) or
        ((T.right is a primitive) and
         (T.left is not a union))
    normalize(T.right)
}
```

Goldfeather et. al. [?] show that the algorithm terminates, generates a tree in normal form and does not add redundant product terms or repeat primitives within a product.

Normalization can add many primitive leaf nodes to a tree with a possibly exponential increase in tree size. In most cases, a large number of the products generated by normalization play no part in the final image, because their primitives do not intersect. A limited amount of geometric information (bounding boxes of primitives) is used to prune CSG trees as they are normalized. Bounding boxes are computed for each operator node using the rules :

1. $\text{Bound}(A \cup B) = \text{Bound}(\text{Bound}(A) \cup \text{Bound}(B))$
2. $\text{Bound}(A \cap B) = \text{Bound}(\text{Bound}(A) \cap \text{Bound}(B))$
3. $\text{Bound}(A - B) = \text{Bound}(A)$

Here A and B are arbitrary child nodes. After each step of the normalization algorithm the tree is pruned by applying the following rules to the current node :

© The Eurographics Association 1997

1. $A \cap B \rightarrow \emptyset$, if Bound($A$) does not intersect Bound(B).
2. $A - B \rightarrow A$, if Bound(A) does not intersect Bound(B).

Normalization of the tree allows simplification of the rendering problem. The union of two or more solids can be rendered using the standard depth (Z) buffer hidden surface removal algorithm used by most graphics workstations. The rendering algorithm needs only to render the correct depth and color for each product in the normalized CSG tree and then allow the depth buffer to combine the results for each product.

Each product can be rendered by rendering each visible surface of a primitive and trimming (intersecting or subtracting) the surface with the remaining primitives in the product. The visible surfaces are the front facing surfaces of uncomplemented primitives and the back facing surfaces of complemented primitives. This observation allows a further rewriting of the CSG tree where each product is split into a sum of *partial products*. A convex primitive has one pair of front and back surfaces per pixel. A non-convex primitive may have any number of pairs of front and back surfaces per pixel. A k-convex primitive is defined as one that has at most k pairs of front and back surfaces per pixel from any view point. We shall use the notation A_k to represent a k-convex primitive and A_{fn} to represent the nth front surface (numbered 0 to $k-1$) of primitive A_k and A_{bn} to represent the nth back surface of A_k. In the common case of convex primitives, we shall drop the numerical subscripts. Thus, $A-B$ expands to $(A_f - B) \cup (B_b \cap A)$ in sum of partial products form; while $A_2 - B$ expands to $(A_{f0} - B) \cup (A_{f1} - B) \cup (B_b \cap A_2)$. We call the primitive whose surface is being rendered the *target* primitive of the partial product. The remaining primitives are called *trimming* primitives.

The sum of partial products form again simplifies the rendering problem. It is now reduced to correctly rendering partial products before combining the results with the depth buffer. Additional *difference* pruning may also be carried out when products have been expanded to partial products :

3 $A_b \cap B \rightarrow \emptyset$, if Bound($A$) does not intersect Bound(B).

A partial product is rendered by first rendering the target surface of the partial product. Each pixel in the surface is then classified in parallel against each of the trimming primitives. To be part of the partial product surface, each pixel must be *in* with respect to any uncomplemented primitives and *out* with respect to any complemented ones. Those pixels which do not meet these criteria are trimmed away (colour set to background, depth set to initial value).

© The Eurographics Association 1997

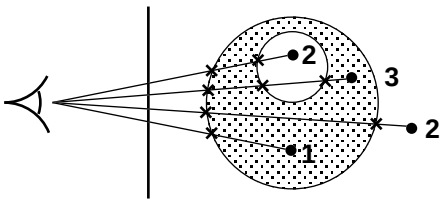


Figure 1: *Classifying per pixel depth values against a primitive*

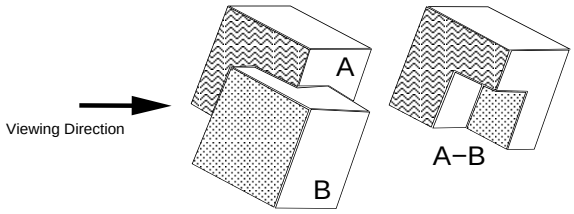


Figure 2: *A simple CSG expression*

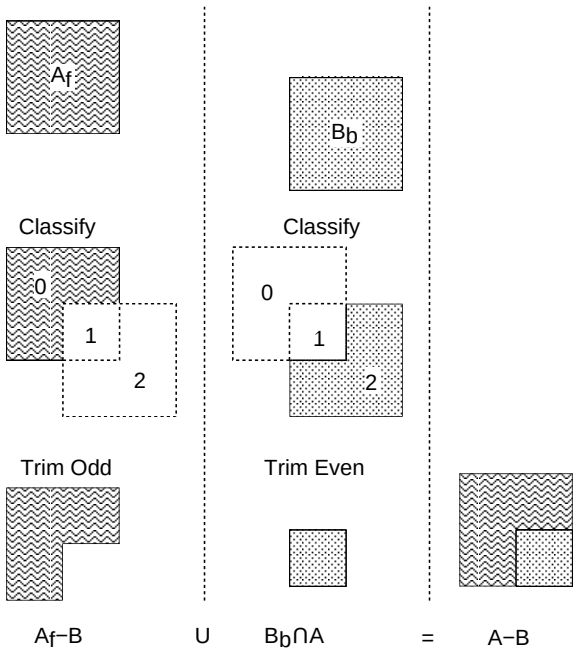


Figure 3: *Rendering figure ?? as two partial products*

Primitives must be formed from closed (possibly nested) facetted shells. Pixels can then be classified against a trimming primitive by counting the number of times a primitive fragment is closer during scan conversion of the primitive's faces. If the result is odd the pixel is *in* with respect to the primitive (figure **??**). Pixels can be classified in parallel by using a 1 bit flag per pixel whose value is toggled whenever scan conversion of a trimming primitive fragment is closer than the pixel's depth value.

Figure **??** illustrates the process for $A-B$ looking along the view direction shown in figure **??**. First, A_f is rendered, classified against B and trimmed (A_f–B). Then B_b is rendered, classified against A and trimmed ($B_b \cap A$). Finally, the two renders are composited together.

Rendering the appropriate surface of a convex primitive is simple as there is only one pair of front and back surfaces per pixel. Most graphics libraries support front and back face culling modes. To render all possible surfaces of an arbitrary k-convex primitive separately requires a $\log_2 k$ bit count per pixel. To render the jth front (or back) facing surface of a primitive, the front (or back) facing surfaces are rendered incrementing the count for each pixel and only enabling writes to the colour and depth buffers for which the count is equal to j.

3. Implementation on a conventional graphics workstation

The algorithm described in section **??** maps naturally onto a hardware architecture which can support two depth buffers, two colour buffers and a stencil buffer. One pair of depth and colour buffers, together with the stencil buffer, are used to render each partial product. The results are then composited into the other pair of buffers. Unfortunately, conventional graphics workstation hardware typically supports only one depth buffer. One approach is to use the hardware provided depth, colour and stencil buffers to render partial products; retrieving the results from the hardware and compositing in local workstation memory. The final result can then be returned direct to the frame buffer. This approach does not make the best use of the workstation hardware. Modern hardware tends to be highly pipelined. Interrupting the pipeline to retrieve results for each partial product will have a considerable performance penalty. In addition, the hardware is typically optimized for flow of data from local memory, through the pipeline and into the frame buffer. Data paths from the frame buffer back to local memory are likely to be slow, especially given the volume of data to be retrieved compared to the compact instructions given to the hardware to draw the primitives. Finally, the compositing operation in local memory will receive no help from the hardware.

Our approach attempts to extract the maximum benefit from any graphics hardware by minimizing the traffic between local memory and the hardware and by making sure that the hardware can be used for all rendering and compositing operations. The idea is to divide the rendering process into two phases — classification and final rendering. Before rendering begins the current depth buffer contents are saved into local memory. We then classify each partial product surface in turn. An extra stencil buffer bit (*accumulator*) per surface stores the results of the classification. During this process updates to the colour buffer are disabled. Once classification is complete, we restore the depth buffer to the saved state and enable updates to the colour buffer. Finally, each partial product surface is rendered again using the stored classification results as a mask (or stencil) to control update of the frame buffer. At the same time the depth buffer acts to composite the pixels which pass the stencil test with those already rendered.

The number of surfaces for which we can perform classification is limited by the depth of the stencil buffer. If the capacity of the stencil buffer is exceeded the surfaces must be processed in multiple passes with the depth buffer saved and restored during each pass. We can reduce the amount of data that needs to be copied by only saving the parts of the depth buffer that will be modified by classification during each pass. The first pass of each frame does not need to save the depth buffer at all as the values are known to be those produced by the initial clear. Instead of restoring, the depth buffer is cleared again. Thus, for simple models rendered at the start of a frame, no depth buffer save and restore is needed at all.

A surface may appear in more than one partial product in the normalized CSG tree. We exploit this by using the same accumulator bit for all partial products with the same surface. Classification results for each partial product are ORed with the current contents of the accumulator.

The stencil bits are partitioned into count bits (S_{count}), a parity bit (S_p) and an accumulator bit (S_a) per surface. $\log_2 k$ count bits are required where k is the maximum convexity of any primitive with a surface being classified in the current pass. The count and parity bits are used independently and may be overlapped. Table **??** shows the number of stencil buffer bits required to classify and render a single surface for primitives of varying convexity. The algorithm requires an absolute minimum of 2 bits for 1-convex and 2-convex primitives, classifying and rendering a single surface in a pass. In practice nearly all primitives used

© The Eurographics Association 1997

Convexity	1	2	3–4	5–8	9–16	17-32	33-64	65–128
S_p	1	1	1	1	1	1	1	1
S_{count}	0	1	2	3	4	5	6	7
S_p and S_{count}	1	1	2	3	4	5	6	7
With 1 accumulator (S_0)	2	2	3	4	5	6	7	8
With 3 accumulators ($S_{0..2}$)	4	4	5	6	7	8	9	10
With 7 accumulators ($S_{0..6}$)	8	8	9	10	11	12	13	14

Table 1: *Stencil buffer usage with primitive convexity*

in pure CSG trees are 1-convex. With 8 stencil bits the algorithm can render from 7 1-convex primitives, to 1 surface of a 128-convex primitive, in a single pass.

Partial products are gathered into groups such that all the partial products in a group can be classified and rendered in one pass. The capacity of a group is defined as the number of different target surfaces that partial products in the group may contain. Capacity is dependent on the stencil buffer depth and the greatest convexity of any of the target primitives in the group (table **??**). Groups are formed by adding partial products in ascending order of target primitive convexity. Once one partial product with a particular target surface is added, all others with the same target surface can be added without using any extra capacity. Adding a partial product with a higher convexity than any already in the group will reduce the group capacity. If there is insufficient capacity to add the minimum convexity remaining partial product, a new group must be started.

Each group is processed in a separate pass in which all target surface primitives are classified and then rendered. Frame buffer wide operations are limited to areas defined by the projection of the bounding box of the current group or partial product. We present pseudo-code for the complete rendering process below. The procedures "glPrim(prim, tests, buffers, ops, pops)" and "glSet(value, tests, buffer, ops, pops)" should be provided by the graphics library. The first renders (scan converts) a primitive where "tests" are the tests performed at each pixel to determine if it can be updated, "buffers" specifies the set of buffers enabled for writing if the "tests" pass (where C is colour, Z is depth and S is stencil), "ops" are operations performed on the stencil bits at each pixel in the primitive, and "pops" are operations to be performed on the stencil bits at each pixel only if "tests" pass. The second procedure is similar but attempts to globally set values for all pixels. Iris GL [?] and OpenGL [?] are two graphics libraries which provide equivalents to the glPrim and glSet procedures described here. We use the symbol Z_P to denote the depth value at a pixel due to the scan conversion of a primitive, P. Hence, "$Z_P < Z$" is the familiar Z buffer hidden surface removal test. We use Z_f to represent the furthest possible depth value.

```
glSet(0, ALWAYS, S, ∅, ∅)
glSet("far", ALWAYS, Z, ∅, ∅)
for first group G {
    classify(G)
    glSet(Z_f, ALWAYS, Z, ∅, ∅)
    renderGroup(G)
} for each subsequent group G {
      save depth buffer
      glSet(Z_f, ALWAYS, Z, ∅, ∅)
      classify(G)
      restore depth buffer
      renderGroup(G)
}

proc classify(G : group)
{
      a = 0
      for each target surface B in G {
          for each partial product R {
              renderSurface(B)
              for each trimming primitive P in R {
                  trim(P)
              }
              glSet(1, S_a = 0 & Z ≠ Z_f, S_a, ∅, ∅)
              glSet(Z_f, ALWAYS, Z, ∅, ∅)
          }
          a = a + 1
      }
}

proc renderGroup(G : group)
{
      a = 0
      for each target primitive P in G {
          glPrim(P, S_a = 1 & Z_P < Z, C & Z, ∅, ∅)
          glSet(0, ALWAYS, S_a, ∅, ∅)
          a = a + 1
      }
}
```

© The Eurographics Association 1997

Capacity		Maximum Target Primitive Convexity							
		1	2	3–4	5–8	9–16	17-32	33-64	65–128
Stencil Buffer Depth	2	1	1	-	-	-	-	-	-
	3	2	2	1	-	-	-	-	-
	4	3	3	2	1	-	-	-	-
	5	4	4	3	2	1	-	-	-
	6	5	5	4	3	2	1	-	-
	7	6	6	5	4	3	2	1	-
	8	7	7	6	5	4	3	2	1

Table 2: *Group Capacity*

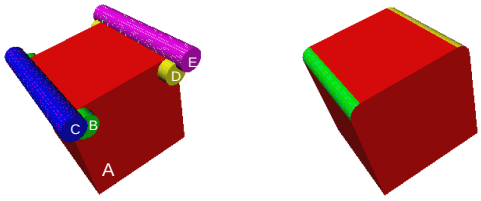


Figure 4: *(a) Primitives, (b) Rendering* $(A \cap B \cup A - C) \cap (A \cap D \cup A - E)$

```
proc renderSurface(B : surface)
{
      P = target primitive containing B
      n = surface number of B
      k = convexity of P
      if P is uncomplemented {
        enable back face culling
      } else {
        enable front face culling
      }
      if k = 1 {
        glPrim(P, ALWAYS, Z, ∅, ∅)
      } else  {
        glPrim(P, S_count = n, Z, inc S_count, ∅)
        glSet(0, ALWAYS, S_count, ∅, ∅)
      }
}

proc trim(P : primitive)
{
      glPrim(P, Z_P < Z, ∅, ∅, toggle S_p)
      if P is uncomplemented {
        glSet(Z_f, S_p = 0 , Z, ∅, ∅)
      } else {
        glSet(Z_f, S_p = 1 , Z, ∅, ∅)
      }
      glSet(0, ALWAYS, S_p, ∅, ∅)
}
```

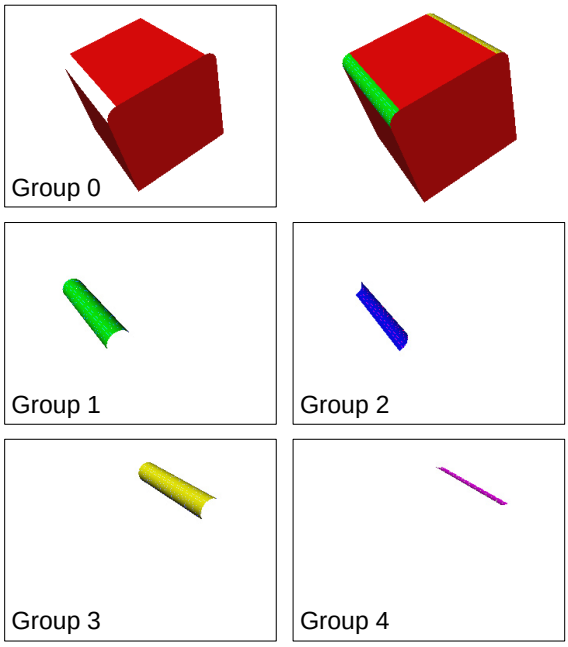


Figure 5: *Rendering each product group separately*

Figure **??** shows five primitives and a rendered CSG tree of the primitives. The expression $((A \cap B) \cup (A - C)) \cap ((A \cap D) \cup (A - E))$ normalizes to $(A \cap B \cap D) \cup (A \cap D - C) \cup (A \cap B - E) \cup (A - C - E)$. Expanding to partial products and grouping gives :

0: $(A_f \cap B \cap D) \cup (A_f \cap D - C) \cup (A_f \cap B - E) \cup (A_f - C - E)$
1: $(B_f \cap A \cap D) \cup (B_f \cap A - E)$
2: $(C_b \cap A \cap D) \cup (C_b \cap A - E)$
3: $(D_f \cap A \cap B) \cup (D_f \cap A - C)$
4: $(E_b \cap A \cap B) \cup (E_b \cap A - C)$

Figure **??** shows the result of rendering each product group separately. Product groups 2 and 4 are not

© The Eurographics Association 1997

visible in the combined image as they are behind the surfaces from groups 1 and 3.

4. Clipping planes and half spaces

Interactive inspection of solid models is aided by means of clipping planes which can help reveal internal structure. After a clipping plane has been defined and activated all subsequently rendered geometry is clipped against the plane and the parts on the *out* side discarded. The rendering of solids as closed shells means that clipping will erroneously reveal the interior of a shell when a portion of the shell is clipped away. Rossignac, Megahed and Schneider [?] describe a stencil buffer based technique for "capping" shells where they intersect a clipping plane. Their algorithm will also highlight interferences (intersections) between solids on the clipping plane.

Clipping a solid and then capping is equivalent to intersection with a half space. We can trivially render an intersection between a solid S and a halfspace H by constructing a convex polygonal primitive P where one face lies on the plane defining H and has edges which do not intersect the bounding box of S. The other faces of P should not intersect S at all. Rendering $S \cap P$ is equivalent to rendering the solid defined by $S \cap H$.

Rossignac, Megahed and Schneider's [?] capping algorithm can be easily integrated with our algorithm to make use of auxiliary clipping planes in rendering CSG trees involving halfspaces. As a halfspace is infinite we assume that it will always be intersected with a finite primitive in any CSG expression. Note that $S - H$ is equivalent to $S \cap \overline{H}$ where $\overline{H}$ is simply H with the normal of the halfspace defining plane reversed.

A halfspace acts as a trimming primitive by activating a clipping plane for the halfspace during the rendering of the target primitive. The stencil buffer is unused. The set of halfspaces in a product can be considered as a 1-convex target primitive. Its surface can be rendered by rendering the defining plane (or rather a sufficiently large polygon lying on the plane) of each halfspace while clipping planes are active for each of the other halfspaces. Each clipping plane is deactivated while it is being rendered to prevent it from clipping itself.

```
proc render(H : halfspace set)
{
    for each defining plane P of H {
        Activate clipping plane defined by P
    }
    for each front facing defining plane P of H {
        Deactivate clipping plane defined by P
        renderPlane(P)
        Activate clipping plane defined by P
    }
    for each defining plane P of H {
        Deactivate clipping plane defined by P
    }
}
```

This approach has three advantages over rendering halfspaces as normal primitives. Firstly, the halfspace set only has to be rendered as a target primitive, all trimming by halfspaces uses the clipping planes. Secondly, each target primitive is clipped, reducing the amount of data written to the frame buffer at the cost of the extra geometry processing required by clipping. Thirdly, a solid/halfspace intersection can be correctly rendered using the algorithm for 1-convex solids ($k = 1$), independent of actual primitive convexity.

Rendering a k-convex target primitive using the algorithm for 1-convex solids results in the nearest surface being drawn (with depth buffering active). The nearest surface (*after* clipping) of a concave primitive will be visible in the intersection with a half space. Rendering an arbitrary CSG tree using the 1-convex algorithm will render the result of evaluating the CSG description on the "nearest spans" (nearest front to nearest back facing surface for each pixel) of each primitive. For interactive use the nearest spans are often all we are interested in. If not, then clipping planes may be used to delimit regions of interest within which the nearest spans will be correctly rendered. Thus, a lower cost, reduced quality mode of rendering is also available.

In addition to user defined clipping planes, all geometry is usually clipped to "near" and "far" planes. These planes are perpendicular to the viewing direction. All geometry must be further from the eye position than the near plane and nearer than the far plane. The near and far planes also define the mapping of distances from the eye point to values stored in the depth buffer. Points on the near plane map to the minimum depth buffer value and points on the far plane map to the maximum depth buffer value. The algorithm described in section **??** will fail if any primitive is clipped by either the near or far clipping plane.

In practice the far clipping plane can always be safely positioned beyond the primitives. The near plane is more troublesome. Firstly, it cannot be positioned behind the eye point. Secondly, the resolution of the depth buffer is critically dependent on the position of the near clip plane. It should be positioned as far from the eye point as possible. Consider rendering $A - B$ and positioning the eye in the hole in A formed by subtracting B. Near plane clipping is unavoidable. We can extend our algorithm to cap trimming prim-

© The Eurographics Association 1997

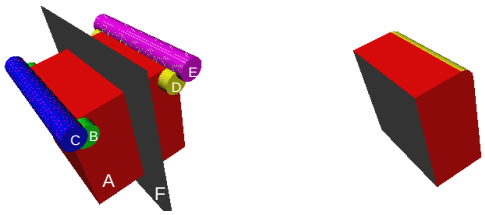


Figure 6: *(a) Primitives, (b) Rendering* $(A \cap B \cup A - C) \cap (A \cap D \cup A - E) \cap F$

itives if they will be subject to near plane clipping. Clipping of target primitives is not a problem unless the eye point is positioned inside the evaluated CSG model.

The trimming primitive is rendered twice while toggling S_p; firstly, with the depth buffer test disabled; secondly, with the depth buffer test enabled. The first render sets the parity bit where capping is required. The second completes the classification as above.

```
proc trim(P : primitive)
{
    glPrim(P, ALWAYS, ∅, ∅, toggle S_p)
    glPrim(P, Z_P < Z, ∅, ∅, toggle S_p)
    if P is uncomplemented {
      glSet(Z_f, S_p = 0 , Z, ∅, ∅)
    } else {
      glSet(Z_f, S_p = 1 , Z, ∅, ∅)
    }
    glSet(0, ALWAYS, S_p, ∅, ∅)
}
```

Figure **??** shows our earlier example intersected with a single clipping plane / half space. The normalized CSG description is $(A \cap B \cap D \cap F) \cup (A \cap D \cap F - C) \cup (A \cap B \cap F - E) \cup (A \cap F - C - E)$.

The normalization and pruning algorithm described in section **??** needs to be extended to cope with halfspace primitives. The extensions required are in the form of additional rules for bounding box generation, normalization and pruning (H is a halfspace) :

Bounding Box Generation

4. Bound$(A \cap H)$ = Bound(A)

Normalization

0. $X - H \rightarrow X \cap \overline{H}$

Pruning

4. $A \cap H \rightarrow \emptyset$, if Bound$(A)$ is outside H.
5. $A \cap H \rightarrow A$, if Bound(A) is inside H.
6. $A \cap H - B \rightarrow A \cap H$, if Bound$(B)$ is outside H.
7. $A_b \cap H \rightarrow A_b$, if Bound$(A)$ is inside H.
8. $H_f - A \rightarrow H_f$, if Bound(A) does not intersect H.

Our earlier example (figure **??**) contains many pruning possibilities. The normalized CSG tree is $(A \cap B \cap D \cap F) \cup (A \cap D \cap F - C) \cup (A \cap B \cap F - E) \cup (A \cap F - C - E)$. Using rule 1 removes the product $A \cap B \cap D \cap F$ as B and D don't intersect. Rule 2 will reduce the products $A \cap D \cap F - C$ and $A \cap B \cap F - E$ to $A \cap D \cap F$ and $A \cap B \cap F$ as the complemented primitives do not intersect the product. Rule 4 removes the product $A \cap B \cap F$, rule 5 reduces $A \cap D \cap F$ to $A \cap D$ and rule 6 reduces $A \cap F - C - E$ to $A \cap F - E$. The normalized and geometric pruned CSG tree is then $(A \cap D \cap F) \cup (A \cap F - E)$. Expanding to partial products gives $(A_f \cap D \cap F) \cup (D_f \cap A \cap F) \cup (F_f \cap A \cap D) \cup (A_f \cap F - E) \cup (F_f \cap A - E) \cup (E_b \cap A \cap F)$. Finally, difference pruning will reduce $E_b \cap A \cap F$ to $E_b \cap A$ (rule 7) and $F_f \cap A - E$ to $F_f \cap A$ (rule 8).

We also prune products against the viewing volume for the current frame and classify trimming primitive bounding boxes against the near clipping plane to determine whether the extra capping step is necessary.

5. Interactive Rendering

We have incorporated our rendering algorithm in a simple, interactive solid modelling system built with standard components. The main framework is provided by the Inventor object-oriented 3D toolkit [?]. A model is represented by a directed acyclic graph of *nodes*. Operations on models, such as rendering or picking, are performed by means of *actions*. The toolkit may be extended by providing user written nodes and actions. Conventional solid modelling operations are provided by the ACIS geometric modeller [?]. ACIS is an object-oriented, boundary representation, solid modelling kernel.

Our modelling system adds new node types to Inventor which support ACIS modelled solids and CSG trees of solids. We also add a new rendering action which uses our stencil buffer CSG display algorithm to render CSG trees described by Inventor node graphs. A CSG evaluate action uses ACIS to fully evaluate a CSG tree allowing the tree to be replaced with a single evaluated solid node. All the standard Inventor interactive tools are available for editing models.

The system supports large CSG trees while maintaining interactive rendering speeds. During display and editing of a large CSG tree, only a small part of the model will be changing at any time. We "cache"

© The Eurographics Association 1997

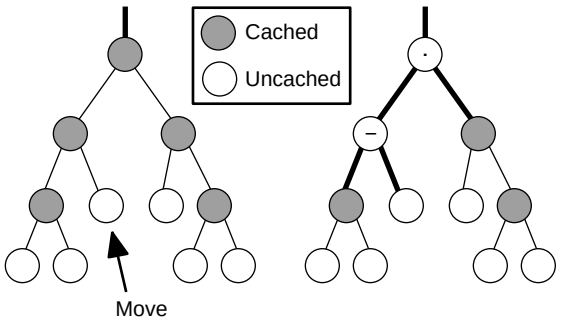


Figure 7: *Direct rendering of a CSG tree with cached geometry : (a) all caches valid, (b) limited direct rendering when a primitive is moved*

fully evaluated geometry obtained from the solid modeller at each internal node in the CSG tree. As caches become invalidated through editing of the model, portions of the tree are rendered directly (see figure **??**), while the cached geometry is re-evaluated in the background (possibly on other workstations in a common network).

Current use of the system follows a common pattern. A user will quickly position and combine primitives using the solid modelling capabilities. During this stage the model is simple enough for the user to envisage the CSG operations required and to position primitives correctly. Figure **??** shows an example model of two intersecting corridors. Firstly, the space occupied by the corridors is modelled using 5 cubes and two cylinders which are unioned together. The corridors are then subtracted from a block. At this point the user wanted to position a skylight through the intersection of the corridors. Unsure of the exact positioning required, or the sort of results possible, the user roughly positioned a cylinder (the hole) and subtracted it from the model. A transparent instance of the primitive is also displayed by the system for reference. A manipulator was then used to drag the hole through the model revealing an unexpected new form. When satisfied with the positioning the hole is "fixed" in position. The fixing process doesn't change the internal representation of the model (it's still a complete CSG tree). It merely hides the apparatus used for interactive manipulation of the hole. The hole can be unfixed at any time and repositioned. This process of rough positioning, boolean combination and precise editing is then repeated.

6. Performance

The time complexity of our algorithm is proportional to the number of rendering operations carried out. We shall consider the rendering of one surface as a single rendering operation. Each pixel oriented "bookkeeping" operation is considered as an equivalent single unit. These operations have a lower geometry overhead than surface rendering but access more pixels. Equivalent functionality could be achieved by performing the bookkeeping operations with a repeated surface render. As in [?], we ignore the negligible normalization and pruning cost. We present the results for our current implementation of the algorithm. For reasons of clarity, some operations are described separately in section **??**, whilst being implemented as a single operation.

Table **??** shows the number of rendering operations required for simple steps within the algorithm. The rendering algorithm is $O((kj)^2)$ for each product where j is the number of primitives in the product and k is the convexity of the primitives. The number of products generated by tree normalization is dependent on the structure of the tree and the geometry of the primitives with a worst case exponential relationship between number of primitives and products. In practice, both we, and Goldfeather et. al. [?], have found that the number of products after pruning is between $O(n)$ and $O(n^2)$ in the total number of primitives. The average product length, j, tends to be small and independent of the total number of primitives. Where long products arise they tend to be of the form $A-B-C-D-E...$ and are susceptible to difference pruning.

Table **??** provides performance statistics for the eight sample models in figure **??**. The images are 500 by 500 pixels and were rendered on a Silicon Graphics 5 span 310/VGXT with a single 33Mhz R3000 processor. The VGXT has an 8 bit stencil buffer. The first part of the table provides statistics on normalization and pruning. We include the number of primitives in the CSG expression, total triangles used to represent the primitives and the number of passes required. The number and average length of partial products produced by normalization with and without pruning are given. The second part of the table provides a breakdown of rendering operations into target rendering, classification & trimming and bookkeeping operations. The third part of the table provides a breakdown of rendering time in seconds; both for rendering operations and depth buffer save/restore time. The depth buffer save/restore time is given for the general case algorithm and for the optimization possible when the model is the first thing rendered in the current frame.

Table **??** shows rendering times together with number of passes required for different stencil buffer sizes. The increases in time are modest because the implementation only saves and restores the areas of the

© The Eurographics Association 1997

Convexity	1-convex ($k=1$)		k-convex	
Clipping	None	Near	None	Near
Classify Target Surface	k	k	$k+1$	$k+1$
Trimming Primitive	$2k+1$	$4k+1$	$2k+1$	$4k+1$
Render Target Surface	k	k	$k+1$	$k+1$

Table 3: *Rendering Operations per Step*

Model	a	b	c	d	e	f	g	h(part)	h(full)
Primitives	2	4	7	31	4	8	2	12	72
Triangles	96	256	408	1532	176	496	1928	8888	5536
Partial Products	2	6	32	34	5	14	3	13	72
Average Length	2	3	4	20.4	2.6	7	2	1.2	2.6
Partial Products (pruned)	2	6	32	34	5	14	3	13	72
Average Length (pruned)	2	3	4	2.7	2.6	3	2	1.2	2.3
Passes	1	1	1	5	1	2	1	1	11
Target Render Ops	2	4	7	30	4	8	5	25	72
Classification & Trimming Ops	4	18	128	92	13	42	8	8	164
Bookkeeping Render Ops	4	18	128	92	13	42	10	10	164
Total Render Ops	10	40	263	214	30	92	23	43	400
Target Time	0.005	0.003	0.038	0.039	0.008	0.017	0.031	0.009	0.118
Classification & Trimming Time	0.026	0.049	0.405	0.268	0.052	0.104	0.033	0.011	0.178
Bookkeeping Time	0.023	0.056	0.180	0.197	0.024	0.108	0.016	0.078	0.098
Save and Restore Time (general)	0.103	0.100	0.114	0.239	0.088	0.217	0.039	0.009	0.236
Save and Restore Time (first)	0.004	0.004	0.001	0.136	0.001	0.111	0.002	0.000	0.214
Total Time (general)	0.165	0.215	0.668	0.772	0.182	0.434	0.126	0.075	0.673
Total Time (first)	0.066	0.119	0.555	0.669	0.095	0.328	0.089	0.067	0.650

Table 4: *Rendering times (seconds) and statistics*

Stencil Bits	8	7	6	5	4	3	2
Model (c)	0.668(1)	0.670(2)	0.701(2)	0.735(2)	0.763(3)	0.782(4)	0.801(7)
Model (d)	0.772(5)	0.779(5)	0.804(6)	0.818(8)	0.837(10)	0.866(15)	0.884(30)
Model (f)	0.434(2)	0.435(2)	0.442(2)	0.426(2)	0.449(3)	0.475(4)	0.504(8)

Table 5: *Rendering time and number of passes with varying stencil size*

© The Eurographics Association 1997

depth buffer that are changed during the classification stage. If less work is done in each pass the changed depth buffer areas typically become smaller. There is scope for further optimization of save and restore as the variations in times for the same number of passes shows. The different stencil buffer size causes a change in the composition of product groups. Placing partial products whose projected bounding boxes overlap into the same product groups will reduce the total area to be saved and restored.

Our algorithm performs particularly well in the sort of situations encountered within our interactive modelling system. Typically there is only ever one "dynamically" rendered CSG expression, usually involving a simple 1-convex "tool" and a more complex "workpiece" (figure **??**(g)). Often we can achieve better performance by ignoring the top most caches of complex workpieces in order to expose more of the CSG tree to pruning. For example, in figure **??**(h) an expression like $(A \cup B \cup C \cup D \cup ..) - X$ can be pruned to $A - X \cup B \cup C \cup D \cup ...$ This can vastly reduce both the number of polygons to be rendered (about 3–5 times as many polygons have to be rendered for A–B compared to $A \cup B$) and the size of the screen area involved in bookkeeping and depth buffer save and restore operations. We provide rendering times for both cached (table **??** h(full)) and uncached cases (table **??** h(part)) of figure **??**(h). The coloured primitives are those that are being "moved", the other geometry can be rendered from caches. The version that makes use of the caches is about 9 times faster than the fully rendered version. However, the triangle count is higher because the cached geometry has a more complex boundary than the original primitives.

Our implementation's performance compares well with that obtained by specialized hardware and pure software solutions. Figure **??**(d) is our version of a model rendered by Goldfeather et. al. [?] on Pixel-Planes 4. They report a total rendering time of 4.02 seconds compared with our time of 0.67 seconds. The VGX architecture machine used for our tests was introduced in 1990 when Pixel-Planes 4 was nearing the end of its lifetime. Pixel-Planes 5 (the most recent machine in the Pixel-Planes series [?]) has performance some 50 times better than Pixel-Planes 4 on a full system with 32 geometry processors and 16 renderers. Such a system would have performance 10 times that of our implementation — at a far greater cost.

Figure **??**(f) is our version of a model rendered by Thibault and Naylor's BSP tree based algorithm [?]. Their total rendering time is 7.2 seconds for a model with 158 polygons on a VAX 8650. Our time is 0.3 seconds for a model with 496 triangles. Our algorithm also scales better with increasing numbers of polygons ($O(kn)$ compared with $O(nlogn)$).

6.1. Other implementations

We have also implemented the algorithm using OpenGL [?] and tested it on our VGXT, a Silicon Graphics R3000 Indigo with starter graphics, and an Indigo^2 Extreme. The algorithm should run under any OpenGL implementation. On the systems we tested performance was comparable to the GL version in all areas except depth buffer save and restore. This operation was about 100 times slower than the GL equivalent. The problem appears to be a combination of poor performance tuning and a specification which requires conversion of the depth buffer values to and from normalized floating point. This problem should be resolved with the release of more mature OpenGL implementations. Single pass renders with the frame start optimization (the common case for our interactive modeller) run at full speed.

7. Conclusion

We have presented an algorithm which directly renders an arbitrary CSG tree and is suitable for use in interactive modelling applications. Unlike Goldfeather et. al. [?], our algorithm requires only a single color buffer, a single depth buffer, a stencil buffer and the ability to save and restore the contents of the depth buffer. It can be implemented on many graphics workstations using existing graphics libraries. Like Rossignac, Megahed and Schneider [?], the algorithm can display cross-sections of solids using clipping planes but is far more flexible. For instance, the algorithm could be used to directly display interferences between solids by rendering the intersection of the solids.

The algorithm has been implemented on an SGI 310/VGXT using the GL graphics library and has been integrated into an experimental modelling system. Performance compares well with specialized hardware and pure software algorithms for complete evaluation and rendering. The algorithm performs particularly well for incremental updates in an interactive modelling environment.

Acknowledgements

This work has been funded by Informatix Inc., Tokyo. Our thanks go to them for their support of the Martin Centre CADLAB over the last four years. Brian Logan, Paul Richens and Simon Schofield have all provided valuable insights and comments; as have the anonymous referees. Paul Richens created the models in figure **??** (g) and (h) using our interactive modeller.

© The Eurographics Association 1997

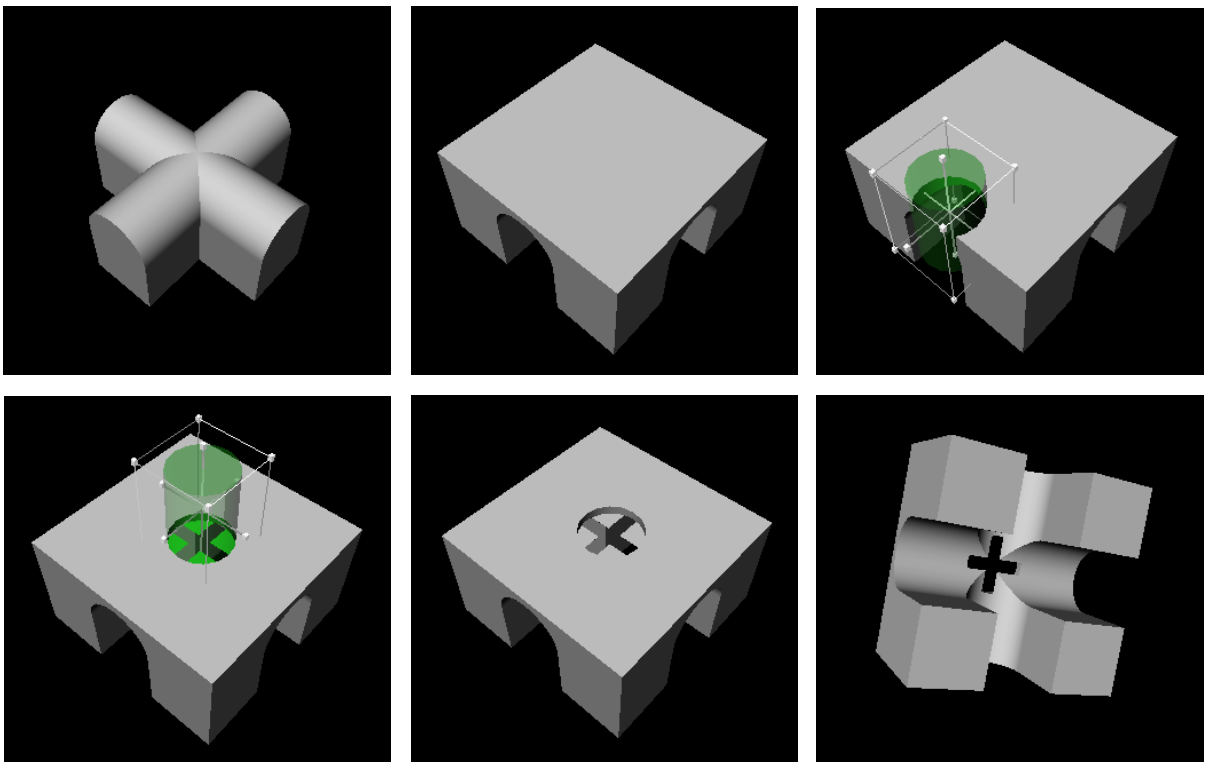


Figure 8: *Dragging a hole though the model reveals an unexpected new form*

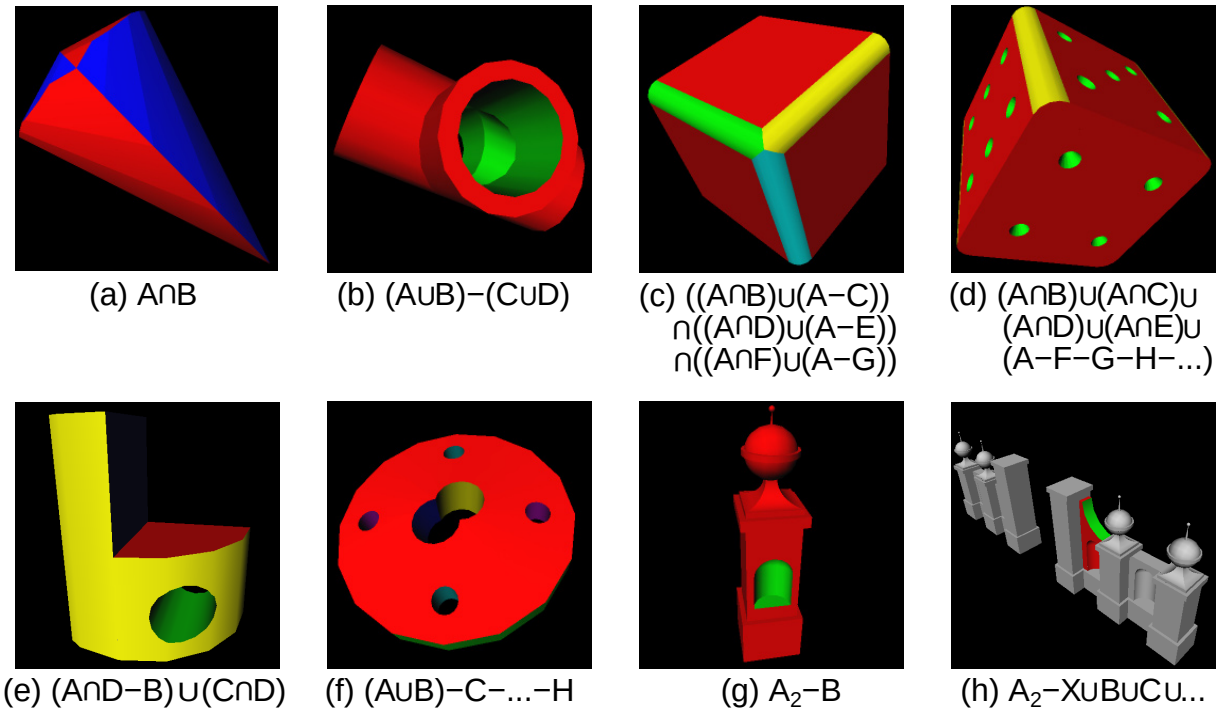


Figure 9: *Images generated by the stencil buffer CSG algorithm*

© The Eurographics Association 1997

Efficient Bump Mapping Hardware (DRAFT COPY)

Mark Peercy
John Airey
Brian Cabral
Silicon Graphics Computer Systems *

Abstract

We present a bump mapping method that requires minimal hardware beyond that necessary for Phong shading. We eliminate the costly per-pixel steps of reconstructing a tangent space and perturbing the interpolated normal vector by a) interpolating vectors that have been transformed into tangent space at polygon vertices and b) storing a precomputed, perturbed normal map as a texture. The savings represents up to a factor of two in hardware or time compared to a straightforward implementation of bump mapping.

CR categories and subject descriptors: I.3.3 [Computer Graphics]: Picture/Image generation; I.3.7 [Image Processing]: Enhancement

Keywords: hardware, shading, bump mapping, texture mapping.

1 INTRODUCTION

Shading calculations in commercially available graphics systems have been limited to lighting at the vertices of a set of polygons, with the resultant colors interpolated and composited with a texture. The drawbacks of Gouraud interpolation [9] are well known and include diffused, crawling highlights and mach banding. The use of this method is motivated primarily by the relatively large cost of the lighting computation. When done at the vertices, this cost is amortized over the interiors of polygons.

The division of a computation into per-vertex and per-pixel components is a general strategy in hardware graphics acceleration [1]. Commonly, the vertex computations are performed in a general floating point processor or cpu, while the per-pixel computations are in special purpose, fixed point hardware. The division is a function of cost versus the general applicability, in terms of quality and speed, of a feature. Naturally, the advance of processor and application-specific integrated circuit technology has an impact on the choice.

Because the per-vertex computations are done in a general processor, the cost of a new feature tends to be dominated by additional per-pixel hardware. If this feature has a very specific application, the extra hardware is hard to justify because it lays idle in applications that do not leverage it. And in low-end or game systems, where every transistor counts, additional rasterization hardware is particularly expensive. An alternative to extra hardware is the reuse of existing hardware, but this option necessarily runs much slower.

* {peercy,airey,cabral}@sgi.com
2011 N. Shoreline Boulevard
Mountain View, California 94043-1389

Shading quality can be increased dramatically with Phong shading [13], which interpolates and normalizes vertex normal vectors at each pixel. Light and halfangle vectors are computed directly in world space or interpolated, either of which requires their normalization for a local viewer and light. Figure 1 shows rasterization

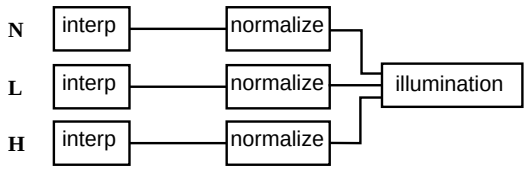


Figure 1. One implementation of Phong shading hardware.

hardware for one implementation of Phong shading, upon which we base this discussion.[1] This adds significant cost to rasterization hardware. However higher quality lighting is almost universally desired in three-dimensional graphics applications, and advancing semiconductor technology is making Phong shading hardware more practical. We take Phong shading and texture mapping hardware as a prerequisite for bump mapping, assuming they will be standard in graphics hardware in the future.

Bump mapping [3] is a technique used in advanced shading applications for simulating the effect of light reflecting from small perturbations across a surface. A single component texture map, $f(u,v)$, is interpreted as a height field that perturbs the surface along its normal vector, $\mathbf{N} = (\mathbf{P}_u \times \mathbf{P}_v)/|(\mathbf{P}_u \times \mathbf{P}_v)|$, at each point. Rather than actually changing the surface geometry, however, only the normal vector is modified. From the partial derivatives of the surface position in the u and v parametric directions ($\mathbf{P}_u$ and $\mathbf{P}_v$), and the partial derivatives of the image height field in u and v (f_u and f_v), a perturbed normal vector $\mathbf{N}'$ is given by [3]:

$$\text{where} \quad \mathbf{N}' = ((\mathbf{P}_u \times \mathbf{P}_v) + \mathbf{D})/|(\mathbf{P}_u \times \mathbf{P}_v) + \mathbf{D}| \qquad (1)$$

$$\mathbf{D} = -f_u(\mathbf{P}_v \times \mathbf{N}) - f_v(\mathbf{N} \times \mathbf{P}_u) \qquad (2)$$

In these equations, $\mathbf{P}_u$ and $\mathbf{P}_v$ are not normalized. As Blinn points out [3], this causes the bump heights to be a function of the surface scale because $\mathbf{P}_u \times \mathbf{P}_v$ changes at a different rate than $\mathbf{D}$. If the surface scale is doubled, the bump heights are halved. This dependence on the surface often is an undesirable feature, and Blinn suggests one way to enforce a constant bump height.

A full implementation of these equations in a rasterizer is impractical, so the computation is divided among a preprocessing step, per-vertex, and per-pixel calculations. A natural method to implement bump mapping in hardware, and one that is planned for a high-end graphics workstation [6], is to compute $\mathbf{P}_u \times \mathbf{P}_v$, $\mathbf{P}_v \times \mathbf{N}$, and $\mathbf{N} \times \mathbf{P}_u$ at polygon vertices and interpolate them to polygon interiors. The perturbed normal vector is computed and normalized as in Equation 1, with f_u and f_v read from a texture map. The resulting normal vector is used in an illumination model.

The hardware for this method is shown in Figure 2. Because $\mathbf{P}_u$

[1] Several different implementations of Phong shading have been suggested [11][10][4][5][7][2] with their own costs and benefits. Our bump mapping algorithm can leverage many variations, and we use this form as well as Blinn's introduction of the halfangle vector for clarity.

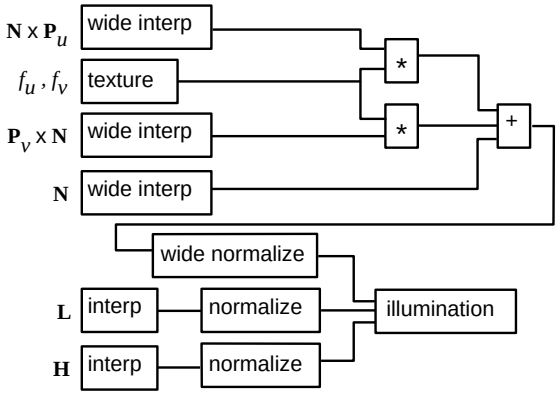


Figure 2. A suggested implementation of bump mapping hardware.

and $\mathbf{P}_v$ are unbounded, the three interpolators, the vector addition, vector scaling, and normalization must have much greater range and precision than those needed for bounded vectors. These requirements are noted in the figure. One approximation to this implementation has been been proposed [8], where $\mathbf{P}_v \times \mathbf{N}$ and $\mathbf{N} \times \mathbf{P}_u$ are held constant across a polygon. While avoiding their interpolation, this approximation is known to have artifacts [8].

We present an implementation of bump mapping that leverages Phong shading hardware at full speed, eliminating either a large investment in special purpose hardware or a slowdown during bump mapping. The principal idea is to transform the bump mapping computation into a different reference frame. Because illumination models are a function of vector operations (such as the dot product) between the perturbed normal vector and other vectors (such as the light and halfangle), they can be computed relative to any frame. We are able to push portions of the bump mapping computation into a preprocess or the per-vertex processor and out of the rasterizer. As a result, minimal hardware is added to a Phong shading circuit.

2 OUR BUMP-MAPPING ALGORITHM

We proceed by recognizing that the original bump mapping approximation [3] assumes a surface is locally flat at each point. The perturbation is, therefore, a function only of the local tangent space. We define this space by the normal vector, $\mathbf{N}$, a tangent vector, $\mathbf{T} = \mathbf{P}_u/|\mathbf{P}_u|$, and a binormal vector, $\mathbf{B} = (\mathbf{N} \times \mathbf{T})$. $\mathbf{T}$, $\mathbf{B}$, and $\mathbf{N}$ form an orthonormal coordinate system in which we perform the bump mapping. In this space, the perturbed normal vector is (see appendix):

$$\mathbf{N}'_{TS} = (a, b, c)/\sqrt{a^2 + b^2 + c^2} \tag{3}$$

$$a = -f_u(\mathbf{B} \cdot \mathbf{P}_v) \tag{4}$$

$$b = -(f_v|\mathbf{P}_u| - f_u(\mathbf{T} \cdot \mathbf{P}_v)) \tag{5}$$

$$c = |\mathbf{P}_u \times \mathbf{P}_v| \tag{6}$$

The coefficients a, b, and c are a function of the surface itself (via $\mathbf{P}_u$ and $\mathbf{P}_v$) and the height field (via f_u and f_v). Provided that the bump map is fixed to a surface, the coefficients can be precomputed for that surface at each point of the height field and stored as a texture map (we discuss approximations that relax the surface dependence below). The texel components lie in the range -1 to 1.

The texture map containing the perturbed normal vector is filtered as a simple texture using, for instance, tri-linear mipmap filtering. The texels in the coarser levels of detail can be computed by filtering finer levels of detail and renormalizing or by filtering the height field and computing the texels directly from Equations 3-6. It is well known that this filtering step tends to average out the bumps at large minifications, leading to artifacts at silhouette edges. Proper filtering of bump maps requires computing the reflected radiance over all bumps contributing to a single pixel, an option that is not practical for hardware systems. It should also be noted that, after mipmap interpolation, the texture will not be normalized, so we must normalize it prior to lighting.

For the illumination calculation to proceed properly, we transform the light and halfangle vectors into tangent space via a 3×3 matrix whose columns are $\mathbf{T}$, $\mathbf{B}$, and $\mathbf{N}$. For instance, the light vector, $\mathbf{L}$, is transformed by

$$\mathbf{L}_{TS} = \mathbf{L} \begin{pmatrix} \mathbf{T} & \mathbf{B} & \mathbf{N} \\ \downarrow & \downarrow & \downarrow \end{pmatrix} \tag{7}$$

Now the diffuse term in the illumination model can be computed from the perturbed normal vector from the texture map and the transformed light: $\mathbf{N}'_{TS} \cdot \mathbf{L}_{TS}$. The same consideration holds for the other terms in the illumination model.

The transformations of the light and halfangle vectors should be performed at every pixel; however, if the change of the local tangent space across a polygon is small, a good approximation can be obtained by transforming the vectors only at the polygon vertices. They are then interpolated and normalized in the polygon interiors. This is frequently a good assumption because tangent space changes rapidly in areas of high surface curvature, and an application will need to tessellate the surfaces more finely in those regions to reduce geometric faceting.

This transformation is, in spirit, the same as one proposed by Kuijk and Blake to reduce the hardware required for Phong shading [11]. Rather than specifying a tangent and binormal explicitly, they rotate the reference frames at polygon vertices to orient all normal vectors in the same direction (such as $(0, 0, 1)$). In this space, they no longer interpolate the normal vector (an approximation akin to ours that tangent space changes slowly). If the bump map is identically zero, we too can avoid an interpolation and normalization, and we will have a result similar to their approximation. It should be noted that the highlight in this case is slightly different than that obtained by the Phong circuit of Figure 1, yet it is still phenomenologically reasonable.

The rasterization hardware required for our bump mapping algorithm is shown in Figure 3; by adding a multiplexer to the Phong shading hardware of Figure 1, both the original Phong shading and bump mapping can be supported. Absent in the implementation of Figure 2, this algorithm requires transforming the light and halfangle vectors into tangent space at each vertex, storing a three-component texture map instead of a two-component map, and having a separate map for each surface. However, it requires only a multiplexer beyond Phong shading, avoids the interpolation of $(\mathbf{P}_v \times \mathbf{N})$ and $(\mathbf{N} \times \mathbf{P}_u)$, the perturbation of the normal vector at each pixel, and the extended range and precision needed for arithmetic on unbounded vectors. Effectively, we have traded per-pixel calculations cast in hardware for per-vertex calculations done in the general geometry processor. If the application is limited by the rasterization, it will run at the same speed with bump mapping as with Phong shading.

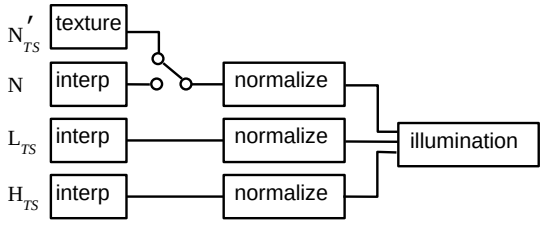


Figure 3. One implementation of our bump mapping algorithm.

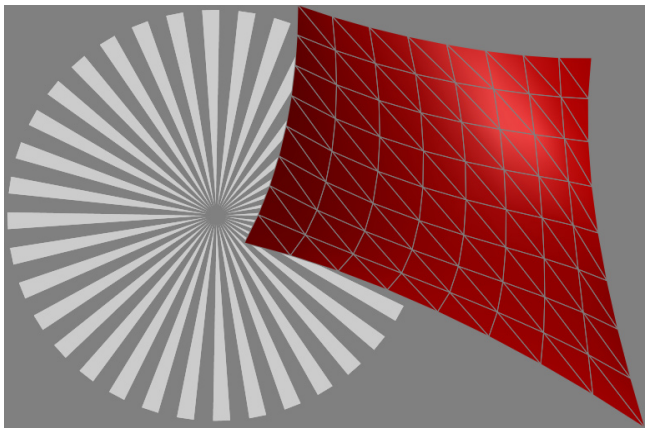


Figure 4.The pinwheel height field is used as a bump map for the tesselated, bicubic surface.

2.1 Object-Space Normal Map

If the texture map is a function of the surface parameterization, another implementation is possible: the lighting model can be computed in object space rather than tangent space. Then, the texture stores the perturbed normal vectors in object space, and the light and halfangle vectors are transformed into object space at the polygon vertices and interpolated. Thus, the matrix transformation applied to the light and halfangle vectors is shared by all vertices, rather than one transformation for each vertex. This implementation keeps the rasterization hardware of Figure 3, significantly reduces the overhead in the geometry processor, and can coexist with the first formulation.

2.2 Removing the surface dependence

The primary drawback of our method is the surface dependence of the texture map. The dependence of the bumps on surface scale is shared with the traditional formulation of bump mapping. Yet in addition, our texture map is a function of the surface, so the height field can not be shared among surfaces with different parameterizations. This is particularly problematic when texture memory is restricted, as in a game system, or during design when a bump map is placed on a new surface interactively.

All of the surface dependencies can be eliminated under the assumption that, locally, the parameterization is the same as a square patch (similar to, yet more restrictive than, the assumption Blinn makes in removing the scale dependence [3]). Then, $\mathbf{P}_u$ and $\mathbf{P}_v$ are orthogonal ($\mathbf{P}_u \cdot \mathbf{P}_v = \mathbf{T} \cdot \mathbf{P}_v = 0$) and equal in magnitude ($|\mathbf{P}_u| = |\mathbf{P}_v|$). To remove the bump dependence on surface scale, we simply choose $|\mathbf{P}_u| = |\mathbf{P}_v| = k$, where k is a constant giving a relative height of the bumps. This, along with the orthogonality condition, reduce Equations 3-6 to

$$\mathbf{N}'_{TS} = (a, b, c)/\sqrt{a^2 + b^2 + c^2} \quad (8)$$

$$a = -kf_u \quad (9)$$

$$b = -kf_v \quad (10)$$

$$c = k^2 \quad (11)$$

The texture map becomes a function only of the height field and not of the surface geometry, so it can be precomputed and used on any surface.

The square patch assumption holds for several important surfaces, such as spheres, tori, surfaces of revolution, and flat rectangles. In addition, the property is highly desirable for general surfaces because the further $\mathbf{P}_u$ and $\mathbf{P}_v$ are from orthogonal and equal in magnitude, the greater the warp in the texture map when applied to a surface. This warping is typically undesirable, and its elimination has been the subject of research [12]. If the surface is already reasonably parameterized or can be reparameterized, the approximation in Equations 8-11 is good.

3 EXAMPLES

Figures 5-7 compare software simulations of the various bump mapping implementations. All of the images, including the height field, have a resolution of 512x512 pixels. The height field, Figure 4, was

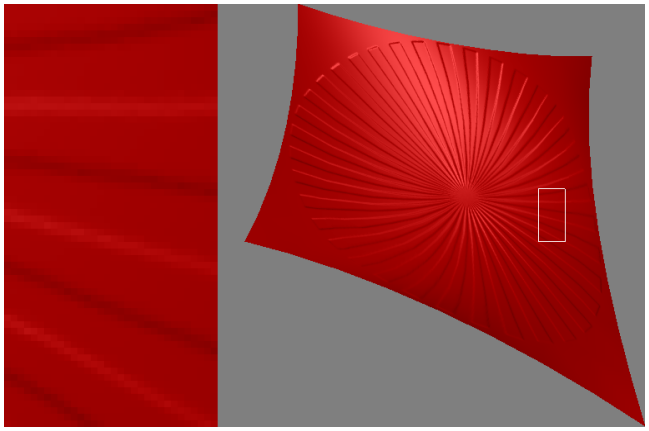


Figure 4. Bump mapping using the hardware implementation shown in Figure 2.

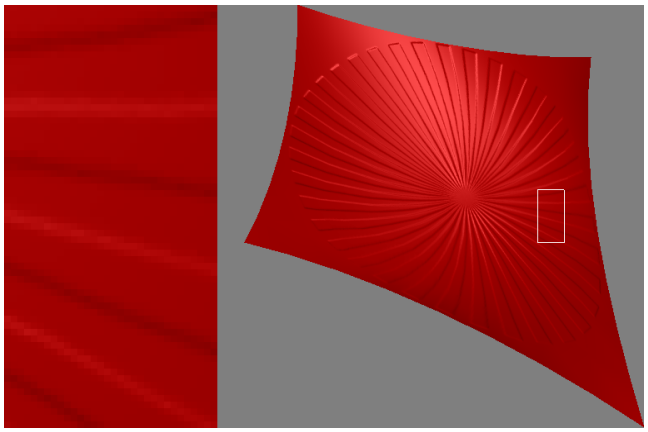


Figure 6.Bump mapping with the hardware in Figure 3, and the texture map from Eqns 3-6.

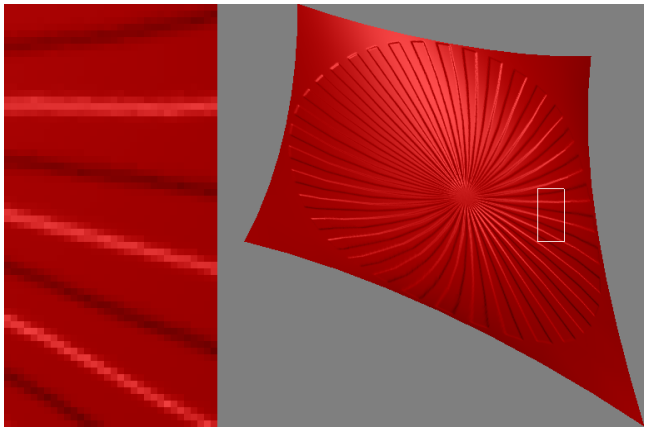


Figure 7.Bump mapping with the hardware in Figure 3, and the texture map from Eqns 8-11.

chosen as a pinwheel to highlight filtering and implementation artifacts, and the surface, Figure 4, was chosen as a highly stretched bicubic patch subdivided into 8x8x2 triangles to ensure that $\mathbf{P}_u$ and $\mathbf{P}_v$ deviate appreciably from orthogonal. The texture maps were filtered with trilinear mipmapping.

Figure 5 shows the image computed from the implementation of bump mapping from Figure 2. The partial derivatives, f_u and f_v, in this texture map and the others were computed with the derivative of a Gaussian covering seven by seven samples.

Figures 6 and 7 show our implementation based on the hardware of Figure 3; they differ only in the texture map that is employed. Figure 6 uses a texture map based on Equations 3-6. Each texel was computed from the analytic values of $\mathbf{P}_u$ and $\mathbf{P}_v$ for the bicubic patch. The difference between this image and Figure 5 is almost imperceptible, even under animation, as can be seen in the enlarged insets. The texture map used in Figure 7 is based on Equations 8-11, where the surface dependence has been removed. Minor differences can be seen in the rendered image compared to Figures 5 and 6; some are visible in the inset. All three implementations have similar filtering qualities and appearance during animation.

4 DISCUSSION

We have presented an implementation of bump mapping that, by transforming the lighting problem into tangent space, avoids any significant new rasterization hardware beyond Phong shading. To summarize our algorithm, we

- precompute a texture of the perturbed normal in tangent space
- transform all shading vectors into tangent space per vertex
- interpolate and renormalize the shading vectors
- fetch and normalize the perturbed normal from the texture
- compute the illumination model with these vectors

Efficiency is gained by moving a portion of the problem to the vertices and away from special purpose bump mapping hardware in the rasterizer; the incremental cost of the per-vertex transformations is amortized over the polygons.

It is important to note that the method of transforming into tangent space for bump mapping is independent of the illumination model, provided the model is a function only of vector operations on the normal. For instance, the original Phong lighting model, with the reflection vector and the view vector for the highlight, can be used instead of the halfangle vector. In this case, the view vector is transformed into tangent space and interpolated rather than the halfangle. As long as all necessary shading vectors for the illumination model are transformed into tangent space and interpolated, lighting is proper.

Our approach is relatively independent of the particular implementation of Phong shading, however it does require the per-pixel illumination model to accept vectors rather than partial illumination results. We have presented a Phong shading circuit where almost no new hardware is required, but other implementations may need extra hardware. For example, if the light and halfangle vectors are computed directly in eye space, interpolators must be added to support our algorithm. The additional cost still will be very small compared to a straightforward implementation.

Phong shading likely will become a standard addition to hardware graphics system because of its general applicability. Our algorithm extends Phong shading in such an effective manner that it is natural to support bump mapping even on the lowest cost Phong shading systems.

5 ACKNOWLEDGEMENTS

This work would not have been possible without help, ideas, conversations and encouragement from Pat Hanrahan, Bob Drebin, Kurt Akeley, Erik Lindholm and Vimal Parikh. Also thanks to the anonymous reviewers who provided good and insightful suggestions.

APPENDIX

Here we derive the perturbed normal vector in tangent space, a reference frame given by tangent, $\mathbf{T} = \mathbf{P}_u/|\mathbf{P}_u|$; binormal, $\mathbf{B} = (\mathbf{N} \times \mathbf{T})$; and normal, $\mathbf{N}$, vectors. $\mathbf{P}_v$ is in the plane of the tangent and binormal, and it can be written:

$$\mathbf{P}_v = (\mathbf{T} \cdot \mathbf{P}_v)\mathbf{T} + (\mathbf{B} \cdot \mathbf{P}_v)\mathbf{B} \quad (12)$$

Therefore

$$\mathbf{P}_v \times \mathbf{N} = (\mathbf{B} \cdot \mathbf{P}_v)\mathbf{T} - (\mathbf{T} \cdot \mathbf{P}_v)\mathbf{B} \quad (13)$$

The normal perturbation (Equation 2) is:

$$\mathbf{D} = -f_u(\mathbf{P}_v \times \mathbf{N}) - f_v|\mathbf{P}_u|\mathbf{B} \quad (14)$$

$$= -f_u(\mathbf{B} \cdot \mathbf{P}_v)\mathbf{T} - (f_v|\mathbf{P}_u| - f_u(\mathbf{T} \cdot \mathbf{P}_v))\mathbf{B} \quad (15)$$

Substituting the expression for $\mathbf{D}$ and $\mathbf{P}_u \times \mathbf{P}_v = |\mathbf{P}_u \times \mathbf{P}_v|\mathbf{N}$ into Equation 1, normalizing, and taking $\mathbf{T}_{TS} = (1, 0, 0)$, $\mathbf{B}_{TS} = (0, 1, 0)$, and $\mathbf{N}_{TS} = (0, 0, 1)$ leads directly to Equations 3-6.

References

[1] AKELEY, K. RealityEngine graphics. In *Computer Graphics (SIGGRAPH '93 Proceedings)* (Aug. 1993), J. T. Kajiya, Ed., vol. 27, pp. 109–116.

[2] BISHOP, G., AND WEIMER, D. M. Fast Phong shading. In *Computer Graphics (SIGGRAPH '86 Proceedings)* (Aug. 1986), D. C. Evans and R. J. Athay, Eds., vol. 20, pp. 103–106.

[3] BLINN, J. F. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)* (Aug. 1978), vol. 12, pp. 286–292.

[4] CLAUSSEN, U. Real time phong shading. In *Fifth Eurographics Workshop on Graphics Hardware* (1989), D. Grimsdale and A. Kaufman, Eds.

[5] CLAUSSEN, U. On reducing the phong shading method. *Computers and Graphics 14*, 1 (1990), 73–81.

[6] COSMAN, M. A., AND GRANGE, R. L. CIG scene realism: The world tomorrow. In *Proceedings of I/ITSEC 1996 on CD-ROM* (1996), p. 628.

[7] DEERING, M. F., WINNER, S., SCHEDIWY, B., DUFFY, C., AND HUNT, N. The triangle processor and normal vector shader: A VLSI system for high performance graphics. In *Computer Graphics (SIGGRAPH '88 Proceedings)* (Aug. 1988), J. Dill, Ed., vol. 22, pp. 21–30.

[8] ERNST, I., JACKEL, D., RUSSELER, H., AND WITTIG, O. Hardware supported bump mapping: A step towards higher quality real-time rendering. In *10th Eurographics Workshop on Graphics Hardware* (1995), pp. 63–70.

[9] GOURAUD, H. Computer display of curved surfaces. *IEEE Trans. Computers C-20*, 6 (1971), 623–629.

[10] JACKEL, D., AND RUSSELER, H. A real time rendering system with normal vector shading. In *9th Eurographics Workshop on Graphics Hardware* (1994), pp. 48–57.

[11] KUIJK, A. A. M., AND BLAKE, E. H. Faster phong shading via angular interpolation. *Computer Graphics Forum 8*, 4 (Dec. 1989), 315–324.

[12] MAILLOT, J., YAHIA, H., AND VERROUST, A. Interactive texture mapping. In *Computer Graphics (SIGGRAPH '93 Proceedings)* (Aug. 1993), J. T. Kajiya, Ed., vol. 27, pp. 27–34.

[13] PHONG, B.-T. Illumination for computer generated pictures. *Communications of the ACM 18*, 6 (June 1975), 311–317.

Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation

Philippe Lacroute
Computer Systems Laboratory
Stanford University

Marc Levoy
Computer Science Department
Stanford University

Abstract

Several existing volume rendering algorithms operate by factoring the viewing transformation into a 3D shear parallel to the data slices, a projection to form an intermediate but distorted image, and a 2D warp to form an undistorted final image. We extend this class of algorithms in three ways. First, we describe a new object-order rendering algorithm based on the factorization that is significantly faster than published algorithms with minimal loss of image quality. Shear-warp factorizations have the property that rows of voxels in the volume are aligned with rows of pixels in the intermediate image. We use this fact to construct a scanline-based algorithm that traverses the volume and the intermediate image in synchrony, taking advantage of the spatial coherence present in both. We use spatial data structures based on run-length encoding for both the volume and the intermediate image. Our implementation running on an SGI Indigo workstation renders a 256^3 voxel medical data set in one second. Our second extension is a shear-warp factorization for perspective viewing transformations, and we show how our rendering algorithm can support this extension. Third, we introduce a data structure for encoding spatial coherence in unclassified volumes (i.e. scalar fields with no precomputed opacity). When combined with our shear-warp rendering algorithm this data structure allows us to classify and render a 256^3 voxel volume in three seconds. The method extends to support mixed volumes and geometry and is parallelizable.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism; I.3.3 [Computer Graphics]: Picture/Image Generation—Display Algorithms.

Additional Keywords: Volume rendering, Coherence, Scientific visualization, Medical imaging.

1 Introduction

Volume rendering is a flexible technique for visualizing scalar fields with widespread applicability in medical imaging and scientific visualization, but its use has been limited because it is computationally expensive. Interactive rendering rates have been reported using large parallel processors [17] [19] and using algorithms that trade off image quality for speed [10] [8], but high-quality images take tens of seconds or minutes to generate on current workstations. In this paper we present a new algorithm which achieves near-interactive rendering rates on a workstation without significantly sacrificing quality.

Many researchers have proposed methods that reduce rendering cost without affecting image quality by exploiting coherence in the data set. These methods rely on spatial data structures that encode the presence or absence of high-opacity voxels so that computation can be omitted in transparent regions of the volume. These data structures are built during a preprocessing step from a *classified* volume: a volume to which an opacity transfer function has been applied. Such spatial data structures include octrees and pyramids [13] [12] [8] [3], k-d trees [18] and distance transforms [23]. Although this type of optimization is data-dependent, researchers have reported that in typical classified volumes 70-95% of the voxels are transparent [12] [18].

Algorithms that use spatial data structures can be divided into two categories according to the order in which the data structures are traversed: image-order or object-order. Image-order algorithms operate by casting rays from each image pixel and processing the voxels along each ray [9]. This processing order has the disadvantage that the spatial data structure must be traversed once for every ray, resulting in redundant computation (e.g. multiple descents of an octree). In contrast, object-order algorithms operate by splatting voxels into the image while streaming through the volume data in storage order [20] [8]. However, this processing order makes it difficult to implement early ray termination, an effective optimization in ray-casting algorithms [12].

In this paper we describe a new algorithm which combines the advantages of image-order and object-order algorithms. The method is based on a factorization of the viewing matrix into a 3D shear parallel to the slices of the volume data, a projection to form a distorted intermediate image, and a 2D warp to produce the final image. Shear-warp factorizations are not new. They have been used to simplify data communication patterns in volume rendering algorithms for SIMD parallel processors [1] [17] and to simplify the generation of paths through a volume in a serial image-order algorithm [22]. The advantage of shear-warp factorizations is that scanlines of the volume data and scanlines of the intermediate image are always aligned. In previous efforts this property has been used to develop SIMD volume rendering algorithms. We exploit the property for a different reason: it allows efficient, synchronized access to data structures that separately encode coherence in the volume and the image.

The factorization also makes efficient, high-quality resampling possible in an object-order algorithm. In our algorithm the resam-

Author's Address: Center for Integrated Systems, Stanford University, Stanford, CA 94305-4070
E-mail: lacroute@weevil.stanford.edu, levoy@cs.stanford.edu
World Wide Web: http://www-graphics.stanford.edu/

Copyright ©1994 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commerical advantage and that new copies bear this notice and the full citation on the first page. Abstracting with credit is permitted.

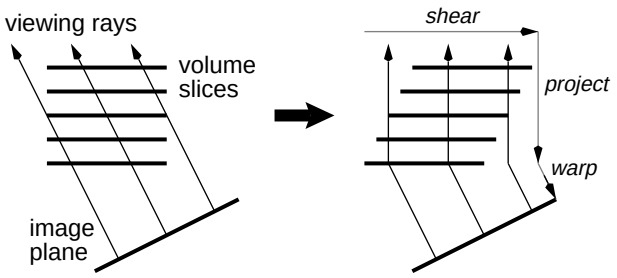


Figure 1: A volume is transformed to sheared object space for a parallel projection by translating each slice. The projection in sheared object space is simple and efficient.

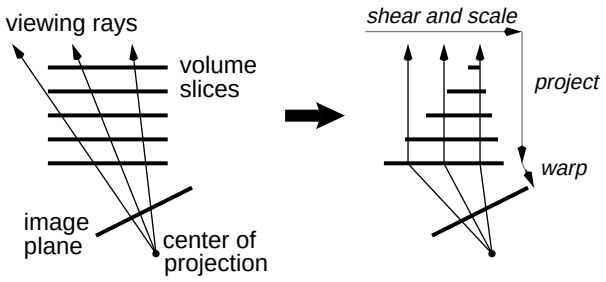


Figure 2: A volume is transformed to sheared object space for a perspective projection by translating and scaling each slice. The projection in sheared object space is again simple and efficient.

pling filter footprint is not view dependent, so the resampling complications of splatting algorithms [20] are avoided. Several other algorithms also use multipass resampling [4] [7] [19], but these methods require three or more resampling steps. Our algorithm requires only two resampling steps for an arbitrary perspective viewing transformation, and the second resampling is an inexpensive 2D warp. The 3D volume is traversed only once.

Our implementation running on an SGI Indigo workstation can render a 256^3 voxel medical data set in one second, a factor of at least five faster than previous algorithms running on comparable hardware. Other than a slight loss due to the two-pass resampling, our algorithm does not trade off quality for speed. This is in contrast to algorithms that subsample the data set and can therefore miss small features [10] [3].

Section 2 of this paper describes the shear-warp factorization and its important mathematical properties. We also describe a new extension of the factorization for perspective projections. Section 3 describes three variants of our volume rendering algorithm. The first algorithm renders classified volumes with a parallel projection using our new coherence optimizations. The second algorithm supports perspective projections. The third algorithm is a fast classification algorithm for rendering unclassified volumes. Previous algorithms that employ spatial data structures require an expensive preprocessing step when the opacity transfer function changes. Our third algorithm uses a classification-independent min-max octree data structure to avoid this step. Section 4 contains our performance results and a discussion of image quality. Finally we conclude and discuss some extensions to the algorithm in Section 5.

2 The Shear-Warp Factorization

The arbitrary nature of the transformation from object space to image space complicates efficient, high-quality filtering and projection in object-order volume rendering algorithms. This problem can be solved by transforming the volume to an intermediate coordinate system for which there is a very simple mapping from the object coordinate system and which allows efficient projection.

We call the intermediate coordinate system "sheared object space" and define it as follows:

> Definition 1: By construction, in sheared object space all viewing rays are parallel to the third coordinate axis.

Figure 1 illustrates the transformation from object space to sheared object space for a parallel projection. We assume the volume is sampled on a rectilinear grid. The horizontal lines in the figure represent slices of the volume data viewed in cross-section. After transformation the volume data has been sheared parallel to the set of slices that is most perpendicular to the viewing direction and the viewing rays are perpendicular to the slices. For a perspective transformation the definition implies that each slice must be scaled as well as sheared as shown schematically in Figure 2.

Definition 1 can be formalized as a set of equations that transform object coordinates into sheared object coordinates. These equations can be written as a factorization of the view transformation matrix M_{view} as follows:

$$M_{\text{view}} = P \cdot S \cdot M_{\text{warp}}$$

P is a permutation matrix which transposes the coordinate system in order to make the z-axis the principal viewing axis. S transforms the volume into sheared object space, and M_{warp} transforms sheared object coordinates into image coordinates. Cameron and Undrill [1] and Schröder and Stoll [17] describe this factorization for the case of rotation matrices. For a general parallel projection S has the form of a shear perpendicular to the z-axis:

$$S_{\text{par}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ s_x & s_y & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where s_x and s_y can be computed from the elements of M_{view}. For perspective projections the transformation to sheared object space is of the form:

$$S_{\text{persp}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ s'_x & s'_y & 1 & s'_w \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

This matrix specifies that to transform a particular slice z_0 of voxel data from object space to sheared object space the slice must be translated by $(z_0 s'_x, z_0 s'_y)$ and then scaled uniformly by $1/(1 + z_0 s'_w)$. The final term of the factorization is a matrix which warps sheared object space into image space:

$$M_{\text{warp}} = S^{-1} \cdot P^{-1} \cdot M_{\text{view}}$$

A simple volume rendering algorithm based on the shear-warp factorization operates as follows (see Figure 3):

1. Transform the volume data to sheared object space by translating and resampling each slice according to S. For perspective transformations, also scale each slice. P specifies which of the three possible slicing directions to use.

2. Composite the resampled slices together in front-to-back order using the "over" operator [15]. This step projects the volume into a 2D intermediate image in sheared object space.

3. Transform the intermediate image to image space by warping it according to M_{warp}. This second resampling step produces the correct final image.

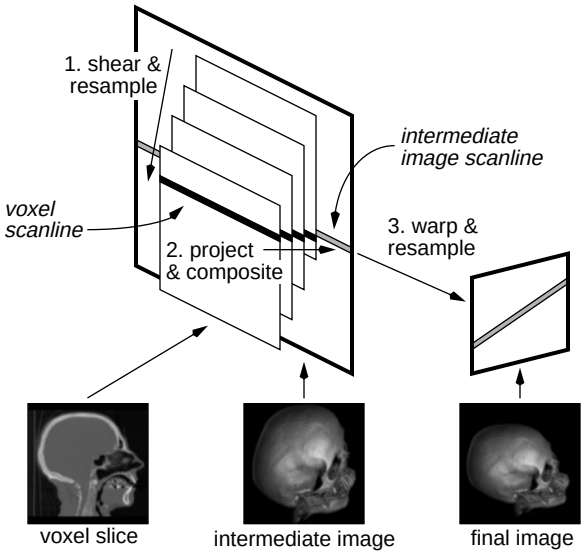


Figure 3: The shear-warp algorithm includes three conceptual steps: shear and resample the volume slices, project resampled voxel scanlines onto intermediate image scanlines, and warp the intermediate image into the final image.

The parallel-projection version of this algorithm was first described by Cameron and Undrill [1]. Our new optimizations are described in the next section.

The projection in sheared object space has several geometric properties that simplify the compositing step of the algorithm:

> Property 1: Scanlines of pixels in the intermediate image are parallel to scanlines of voxels in the volume data.
>
> Property 2: All voxels in a given voxel slice are scaled by the same factor.
>
> Property 3 (parallel projections only): Every voxel slice has the same scale factor, and this factor can be chosen arbitrarily. In particular, we can choose a unity scale factor so that for a given voxel scanline there is a one-to-one mapping between voxels and intermediate-image pixels.

In the next section we make use of these properties.

3 Shear-Warp Algorithms

We have developed three volume rendering algorithms based on the shear-warp factorization. The first algorithm is optimized for parallel projections and assumes that the opacity transfer function does not change between renderings, but the viewing and shading parameters can be modified. The second algorithm supports perspective projections. The third algorithm allows the opacity transfer function to be modified as well as the viewing and shading parameters, with a moderate performance penalty.

3.1 Parallel Projection Rendering Algorithm

Property 1 of the previous section states that voxel scanlines in the sheared volume are aligned with pixel scanlines in the intermediate image, which means that the volume and image data structures can both be traversed in scanline order. Scanline-based coherence data structures are therefore a natural choice. The first data structure we use is a run-length encoding of the voxel scanlines which allows us to take advantage of coherence in the volume by skipping runs of transparent voxels. The encoded scanlines consist of two types

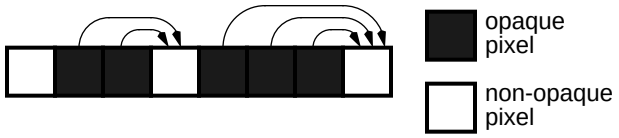


Figure 4: Offsets stored with opaque pixels in the intermediate image allow occluded voxels to be skipped efficiently.

of runs, transparent and non-transparent, defined by a user-specified opacity threshold. Next, to take advantage of coherence in the image, we store with each opaque intermediate image pixel an offset to the next non-opaque pixel in the same scanline (Figure 4). An image pixel is defined to be opaque when its opacity exceeds a user-specified threshold, in which case the corresponding voxels in yet-to-be-processed slices are occluded. The offsets associated with the image pixels are used to skip runs of opaque pixels without examining every pixel. The pixel array and the offsets form a run-length encoding of the intermediate image which is computed on-the-fly during rendering.

These two data structures and Property 1 lead to a fast scanline-based rendering algorithm (Figure 5). By marching through the volume and the image simultaneously in scanline order we reduce addressing arithmetic. By using the run-length encoding of the voxel data to skip voxels which are transparent and the run-length encoding of the image to skip voxels which are occluded, we perform work only for voxels which are both non-transparent and visible.

For voxel runs that are not skipped we use a tightly-coded loop that performs shading, resampling and compositing. Properties 2 and 3 allow us to simplify the resampling step in this loop. Since the transformation applied to each slice of volume data before projection consists only of a translation (no scaling or rotation), the resampling weights are the same for every voxel in a slice (Figure 6). Algorithms which do not use the shear-warp factorization must recompute new weights for every voxel. We use a bilinear interpolation filter and a gather-type convolution (backward projection): two voxel scanlines are traversed simultaneously to compute a single intermediate image scanline at a time. Scatter-type convolution (forward projection) is also possible. We use a lookup-table based system for shading [6]. We also use a lookup table to correct voxel opacity for the current viewing angle since the apparent thickness of a slice of voxels depends on the viewing angle with respect to the orientation of the slice.

The opaque pixel links achieve the same effect as early ray termination in ray-casting algorithms [12]. However, the effectiveness of this optimization depends on coherence of the opaque regions of the image. The runs of opaque pixels are typically large so that many pixels can be skipped at once, minimizing the number of pixels that are examined. The cost of computing the pixel offsets is low because a pixel's offset is updated only when the pixel first becomes

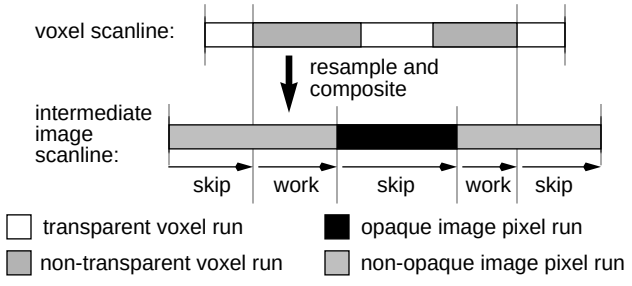


Figure 5: Resampling and compositing are performed by streaming through both the voxels and the intermediate image in scanline order, skipping over voxels which are transparent and pixels which are opaque.

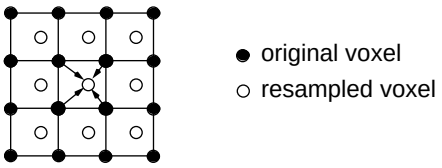


Figure 6: Since each slice of the volume is only translated, every voxel in the slice has the same resampling weights.

opaque.

After the volume has been composited the intermediate image must be warped into the final image. Since the 2D image is small compared to the size of the volume this part of the computation is relatively inexpensive. We use a general-purpose affine image warper with a bilinear filter.

The rendering algorithm described in this section requires a run-length encoded volume which must be constructed in a preprocessing step, but the data structure is view-independent so the cost to compute it can be amortized over many renderings. Three encodings are computed, one for each possible principal viewing direction, so that transposing the volume is never necessary. During rendering one of the three encodings is chosen depending upon the value of the permutation matrix P in the shear-warp factorization. Transparent voxels are not stored, so even with three-fold redundancy the encoded volume is typically much smaller than the original volume (see Section 4.1). Fast computation of the run-length encoded data structure is discussed further at the end of Section 3.3.

In this section we have shown how the shear-warp factorization allows us to combine optimizations based on object coherence and image coherence with very low overhead and simple, high-quality resampling. In the next section we extend these advantages to a perspective volume rendering algorithm.

3.2 Perspective Projection Rendering Algorithm

Most of the work in volume rendering has focused on parallel projections. However, perspective projections provide additional cues for resolving depth ambiguities [14] and are essential to correctly compute occlusions in such applications as a beam's eye view for radiation treatment planning. Perspective projections present a problem because the viewing rays diverge so it is difficult to sample the volume uniformly. Two types of solutions have been proposed for perspective volume rendering using ray-casters: as the distance along a ray increases the ray can be split into multiple rays [14], or each sample point can sample a larger portion of the volume using a mip-map [11] [16]. The object-order splatting algorithm can also handle perspective, but the resampling filter footprint must be recomputed for every voxel [20].

The shear-warp factorization provides a simple and efficient solution to the sampling problem for perspective projections. Each slice of the volume is transformed to sheared object space by a translation and a uniform scale, and the slices are then resampled and composited together. These steps are equivalent to a ray-casting algorithm in which rays are cast to uniformly sample the first slice of volume data, and as each ray hits subsequent (more distant) slices a larger portion of the slice is sampled (Figure 2). The key point is that within each slice the sampling rate is uniform (Property 2 of the factorization), so there is no need to implement a complicated multirate filter.

The perspective algorithm is nearly identical to the parallel projection algorithm. The only difference is that each voxel must be scaled as well as translated during resampling, so more than two voxel scanlines may be traversed simultaneously to produce a given intermediate image scanline and the voxel scanlines may not be traversed at the same rate as the image scanlines. We always choose a factorization of the viewing transformation in which the slice closest to the viewer is scaled by a factor of one so that no slice is ever enlarged. To resample we use a box reconstruction filter and a box low-pass filter, an appropriate combination for both decimation and unity scaling. In the case of unity scaling the two filter widths are identical and their convolution reduces to the bilinear interpolation filter used in the parallel projection algorithm.

The perspective algorithm is more expensive than the parallel projection algorithm because extra time is required to compute resampling weights and because the many-to-one mapping from voxels to pixels complicates the flow of control. Nevertheless, the algorithm is efficient because of the properties of the shear-warp factorization: the volume and the intermediate image are both traversed scanline by scanline, and resampling is accomplished via two simple resampling steps despite the diverging ray problem.

3.3 Fast Classification Algorithm

The previous two algorithms require a preprocessing step to run-length encode the volume based on the opacity transfer function. The preprocessing time is insignificant if the user wishes to generate many images from a single classified volume, but if the user wishes to experiment interactively with the transfer function then the preprocessing step is unacceptably slow. In this section we present a third variation of the shear-warp algorithm that evaluates the opacity transfer function during rendering and is only moderately slower than the previous algorithms.

A run-length encoding of the volume based upon opacity is not an appropriate data structure when the opacity transfer function is not fixed. Instead we apply the algorithms described in Sections 3.1–3.2 to unencoded voxel scanlines, but with a new method to determine which portions of each scanline are non-transparent. We allow the opacity transfer function to be any scalar function of a multi-dimensional scalar domain:

$$\alpha = f(p, q, ...)$$

For example, the opacity might be a function of the scalar field and its gradient magnitude [9]:

$$\alpha = f(d, |\nabla d|)$$

The function f essentially partitions a multi-dimensional feature space into transparent and non-transparent regions, and our goal is to decide quickly which portions of a given scanline contain voxels in the non-transparent regions of the feature space.

We solve this problem with the following recursive algorithm which takes advantage of coherence in both the opacity transfer function and the volume data:

Step 1: For some block of the volume that contains the current scanline, find the extrema of the parameters of the opacity transfer function $(\min(p), \max(p), \min(q), \max(q), ...)$. These extrema bound a rectangular region of the feature space.

Step 2: Determine if the region is transparent, i.e. f evaluated for all parameter points in the region yields only transparent opacities. If so, then discard the scanline since it must be transparent.

Step 3: Subdivide the scanline and repeat this algorithm recursively. If the size of the current scanline portion is below a threshold then render it instead of subdividing.

This algorithm relies on two data structures for efficiency (Figure 7). First, Step 1 uses a precomputed min-max octree [21]. Each octree node contains the extrema of the parameter values for a subcube of the volume. Second, to implement Step 2 of the algorithm we need to integrate the function f over the region of the feature space found in Step 1. If the integral is zero then all voxels must

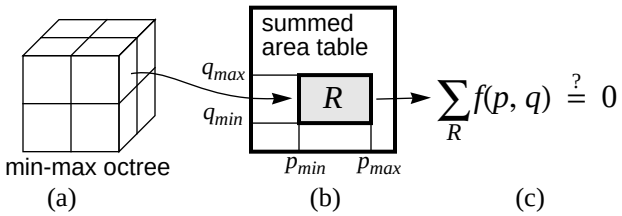


Figure 7: A min-max octree (a) is used to determine the range of the parameters p, q of the opacity transfer function $f(p, q)$ in a subcube of the volume. A summed area table (b) is used to integrate f over that range of p, q. If the integral is zero (c) then the subcube contains only transparent voxels.

be transparent.* This integration can be performed in constant time using a multi-dimensional summed-area table [2] [5]. The voxels themselves are stored in a third data structure, a simple 3D array.

The overall algorithm for rendering unclassified data sets proceeds as follows. The min-max octree is computed at the time the volume is first loaded since the octree is independent of the opacity transfer function and the viewing parameters. Next, just before rendering begins the opacity transfer function is used to compute the summed area table. This computation is inexpensive provided that the domain of the opacity transfer function is not too large. We then use either the parallel projection or the perspective projection rendering algorithm to render voxels from an unencoded 3D voxel array. The array is traversed scanline by scanline. For each scanline we use the octree and the summed area table to determine which portions of the scanline are non-transparent. Voxels in the non-transparent portions are individually classified using a lookup table and rendered as in the previous algorithms. Opaque regions of the image are skipped just as before. Note that voxels that are either transparent or occluded are never classified, which reduces the amount of computation.

The octree traversal and summed area table lookups add overhead to the algorithm which were not present in the previous algorithms. In order to reduce this overhead we save as much computed data as possible for later reuse: an octree node is tested for transparency using the summed area table only the first time it is visited and the result is saved for subsequent traversals, and if two adjacent scanlines intersect the same set of octree nodes then we record this fact and reuse information instead of making multiple traversals.

This rendering algorithm places two restrictions on the opacity transfer function: the parameters of the function must be precomputable for each voxel so that the octree may be precomputed, and the total number of possible argument tuples to the function (the cardinality of the domain) must not be too large since the summed area table must contain one entry for each possible tuple. Context-sensitive segmentation (classification based upon the position and surroundings of a voxel) does not meet these criteria unless the segmentation is entirely precomputed.

The fast-classification algorithm presented here also suffers from a problem common to many object-order algorithms: if the major viewing axis changes then the volume data must be accessed against the stride and performance degrades. Alternatively the 3D array of voxels can be transposed, resulting in a delay during interactive viewing. Unlike the algorithms based on a run-length encoded volume, it is typically not practical to maintain three copies of the unencoded volume since it is much larger than a run-length encoding. It is better to use a small range of viewpoints while modifying the classification function, and then to switch to one of the previous two rendering methods for rendering animation sequences. In fact, the octree and the summed-area table can be used to convert the 3D voxel array into a run-length encoded volume without accessing transparent voxels, leading to a significant time savings (see the "Switch Modes" arrow in Figure 12). Thus the three algorithms fit together well to yield an interactive tool for classifying and viewing volumes.

*The user may choose a non-zero opacity threshold for transparent voxels, in which case a thresholded version of f must be integrated: let $f' = f$ whenever f exceeds the threshold, and $f' = 0$ otherwise.

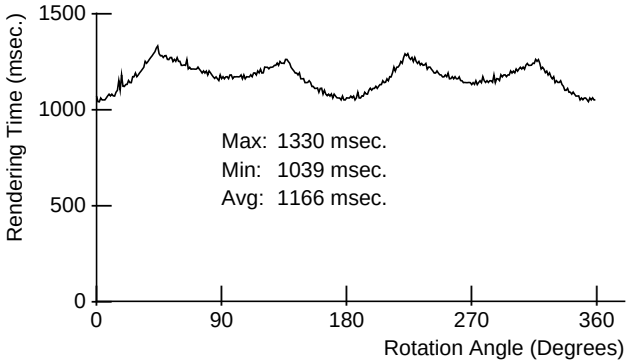


Figure 11: Rendering time for a parallel projection of the head data set as the viewing angle changes.

4 Results

4.1 Speed and Memory

Our performance results for the three algorithms are summarized in Table 1. The "Fast Classification" timings are for the algorithm in Section 3.3 with a parallel projection. The timings were measured on an SGI Indigo R4000 without hardware graphics accelerators. Rendering times include all steps required to render from a new viewpoint, including computation of the shading lookup table, compositing and warping, but the preprocessing step is not included. The "Avg." field in the table is the average time in seconds for rendering 360 frames at one degree angle increments, and the "Min/Max" times are for the best and worst case angles. The "Mem." field gives the size in megabytes of all data structures. For the first two algorithms the size includes the three run-length encodings of the volume, the image data structures and all lookup tables. For the third algorithm the size includes the unencoded volume, the octree, the summed-area table, the image data structures, and the lookup tables. The "brain" data set is an MRI scan of a human head (Figure 8) and the "head" data set is a CT scan of a human head (Figure 9). The "brainsmall" and "headsmall" data sets are decimated versions of the larger volumes.

The timings are nearly independent of image size because this factor affects only the final warp which is relatively insignificant. Rendering time is dependent on viewing angle (Figure 11) because the effectiveness of the coherence optimizations varies with viewpoint and because the size of the intermediate image increases as the rotation angle approaches 45 degrees, so more compositing operations must be performed. For the algorithms described in Sections 3.1–3.2 there is no jump in rendering time when the major viewing axis changes, provided the three run-length encoded copies of the volume fit into real memory simultaneously. Each copy contains four bytes per non-transparent voxel and one byte per run. For the 256x256x226 voxel head data set the three run-length encodings total only 9.8 Mbytes. All of the images were rendered on a workstation with 64 Mbytes of memory. To test the fast classification algorithm (Section 3.3) on the 256^3 data sets we used a workstation with 96 Mbytes of memory.

Figure 12 gives a breakdown of the time required to render the brain data set with a parallel projection using the fast classification algorithm (left branch) and the parallel projection algorithm (right branch). The time required to warp the intermediate image into the final image is typically 10-20% of the total rendering time for the

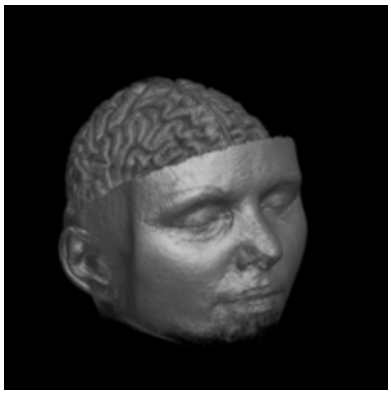


Figure 8: Volume rendering with a parallel projection of an MRI scan of a human brain using the shear-warp algorithm (1.1 sec.).

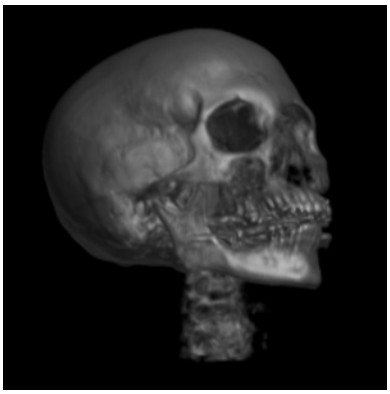


Figure 9: Volume rendering with a parallel projection of a CT scan of a human head oriented at 45 degrees relative to the axes of the volume (1.2 sec.).

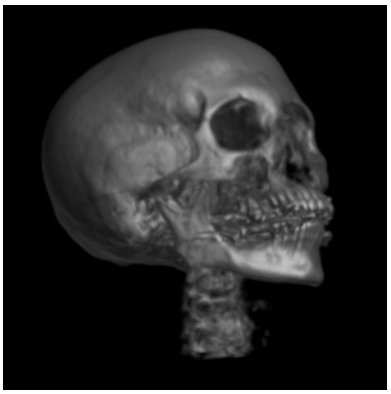


Figure 10: Volume rendering of the same data set as in Figure 9 using a ray-caster [12] for quality comparison (13.8 sec.).

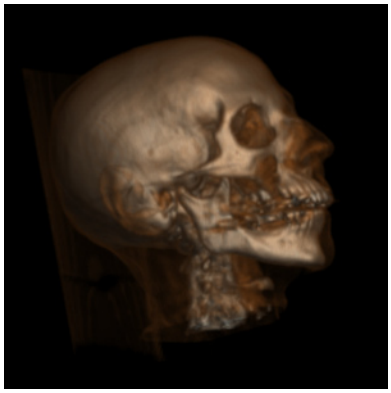


Figure 13: Volume rendering with a parallel projection of the human head data set classified with semitransparent skin (3.0 sec.).

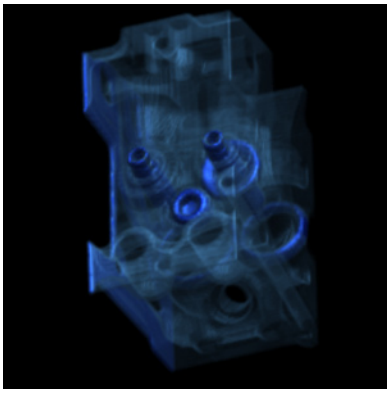


Figure 14: Volume rendering with a parallel projection of an engine block with semitransparent and opaque surfaces (2.3 sec.).

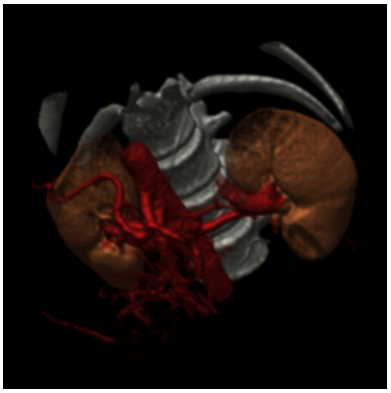


Figure 15: Volume rendering with a parallel projection of a CT scan of a human abdomen (2.2 sec.). The blood vessels contain a radio-opaque dye.

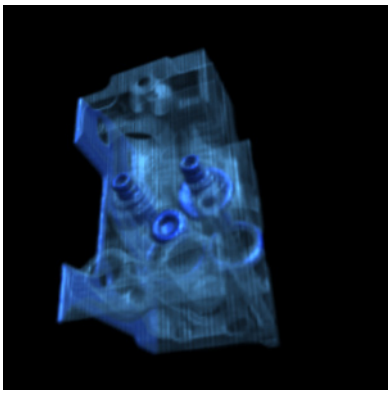
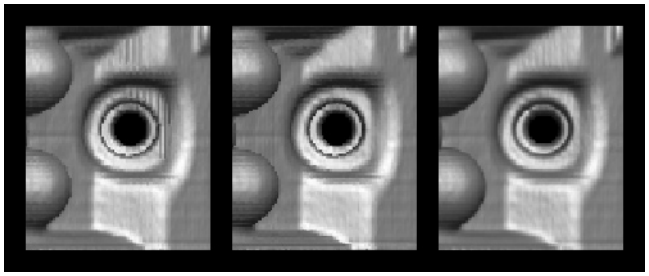


Figure 16: Volume rendering with a perspective projection of the engine data set (3.8 sec.).



(a) (b) (c)

Figure 17: Comparison of image quality with bilinear and trilinear filters for a portion of the engine data set. The images have been enlarged. (a) Bilinear filter with binary classification. (b) Trilinear filter with binary classification. (c) Bilinear filter with smooth classification.

Data set	Size (voxels)	Parallel projection (§3.1)			Perspective projection (§3.2)			Fast classification (§3.3)		
		Avg.	Min/Max	Mem.	Avg.	Min/Max	Mem.	Avg.	Min/Max	Mem.
brainsmall	128x128x109	0.4 s.	0.37–0.48 s.	4 Mb.	1.0 s.	0.84–1.13 s.	4 Mb.	0.7 s.	0.61–0.84 s.	8 Mb.
headsmall	128x128x113	0.4	0.35–0.43	2	0.9	0.82–1.00	2	0.8	0.72–0.87	8
brain	256x256x167	1.1	0.91–1.39	19	3.0	2.44–2.98	19	2.4	1.91–2.91	46
head	256x256x225	1.2	1.04–1.33	13	3.3	2.99–3.68	13	2.8	2.43–3.23	61

Table 1: Rendering time and memory usage on an SGI Indigo workstation. Times are in seconds and include shading, resampling, projection and warping. The fast classification times include rendering with a parallel projection. The "Mem." field is the total size of the data structures used by each algorithm.

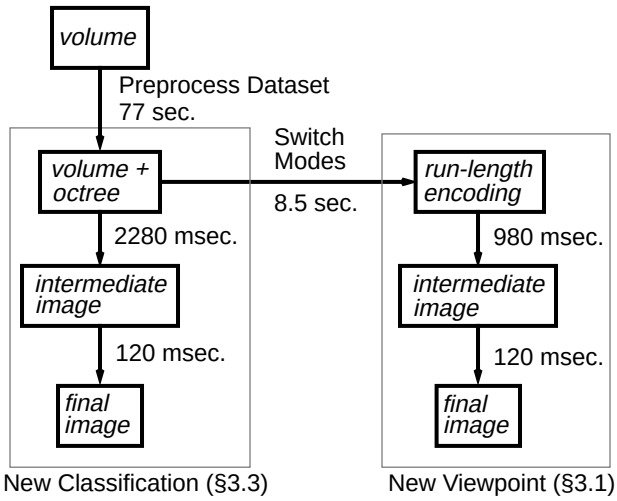


Figure 12: Performance results for each stage of rendering the brain data set with a parallel projection. The left side uses the fast classification algorithm and the right side uses the parallel projection algorithm.

parallel projection algorithm. The "Switch Modes" arrow shows the time required for all three copies of the run-length encoded volume to be computed from the unencoded volume and the min-max octree once the user has settled on an opacity transfer function.

The timings above are for grayscale renderings. Color renderings take roughly twice as long for parallel projections and 1.3x longer for perspective because of the additional resampling required for the two extra color channels. Figure 13 is a color rendering of the head data set classified with semitransparent skin which took 3.0 sec. to render. Figure 14 is a rendering of a 256x256x110 voxel engine block, classified with semi-transparent and opaque surfaces; it took 2.3 sec. to render. Figure 15 is a rendering of a 256x256x159 CT scan of a human abdomen, rendered in 2.2 sec. The blood vessels of the subject contain a radio-opaque dye, and the data set was classified to reveal both the dye and bone surfaces. Figure 16 is a perspective color rendering of the engine data set which took 3.8 sec. to compute.

For comparison purposes we rendered the head data set with a ray-caster that uses early ray termination and a pyramid to exploit object coherence [12]. Because of its lower computational overhead the shear-warp algorithm is more than five times faster for the 128^3 data sets and more than ten times faster for the 256^3 data sets. Our algorithm running on a workstation is competitive with algorithms for massively parallel processors ([17], [19] and others), although the parallel implementations do not rely on coherence optimizations and therefore their performance results are not data dependent as ours are.

Our experiments show that the running time of the algorithms in Sections 3.1–3.2 is proportional to the number of voxels which are resampled and composited. This number is small either if a significant fraction of the voxels are transparent or if the average voxel opacity is high. In the latter case the image quickly becomes opaque and the remaining voxels are skipped. For the data sets and classification functions we have tried roughly n^2 voxels are both non-transparent and visible, so we observe $O(n^2)$ performance as shown in Table 1: an eight-fold increase in the number of voxels leads to only a four-fold increase in time for the compositing stage and just under a four-fold increase in overall rendering time. For our rendering of the head data set 5% of the voxels are non-transparent, and for the brain data set 11% of the voxels are non-transparent. Degraded performance can be expected if a substantial fraction of the classified volume has low but non-transparent opacity, but in our experience such classification functions are less useful.

4.2 Image Quality

Figure 10 is a volume rendering of the same data set as in Figure 9, but produced by a ray-caster using trilinear interpolation [12]. The two images are virtually identical.

Nevertheless, there are two potential quality problems associated with the shear-warp algorithm. First, the algorithm involves two resampling steps: each slice is resampled during compositing, and the intermediate image is resampled during the final warp. Multiple resampling steps can potentially cause blurring and loss of detail. However even in the high-detail regions of Figure 9 this effect is not noticeable.

The second potential problem is that the shear-warp algorithm uses a 2D rather than a 3D reconstruction filter to resample the volume data. The bilinear filter used for resampling is a first-order filter in the plane of a voxel slice, but it is a zero-order (nearest-neighbor) filter in the direction orthogonal to the slice. Artifacts are likely to appear if the opacity or color attributes of the volume contain very high frequencies (although if the frequencies exceed the Nyquist rate then perfect reconstruction is impossible).

Figure 17 shows a case where a trilinear interpolation filter outperforms a bilinear filter. The left-most image is a rendering by the shear-warp algorithm of a portion of the engine data set which has been classified with extremely sharp ramps to produce high frequencies in the volume's opacity. The viewing angle is set to 45 degrees relative to the slices of the data set—the worst case—and aliasing is apparent. For comparison, the middle image is a rendering produced with a ray-caster using trilinear interpolation and otherwise identical rendering parameters; here there is virtually no aliasing. However, by using a smoother opacity transfer function these reconstruction artifacts can be reduced. The right-most image is a rendering using the shear-warp algorithm and a less-extreme opacity transfer function. Here the aliasing is barely noticeable because the high frequencies in the scalar field have effectively been low-pass filtered by the transfer function. In practice, as long as the opacity transfer function is not a binary classification the bilinear filter produces good results.

5 Conclusion

The shear-warp factorization allows us to implement coherence optimizations for both the volume data and the image with low computational overhead because both data structures can be traversed simultaneously in scanline order. The algorithm is flexible enough to

accommodate a wide range of user-defined shading models and can handle perspective projections. We have also presented a variant of the algorithm that does not assume a fixed opacity transfer function. The result is an algorithm which produces high-quality renderings of a 256^3 volume in roughly one second on a workstation with no specialized hardware.

We are currently extending our rendering algorithm to support data sets containing both geometry and volume data. We have also found that the shear-warp algorithms parallelize naturally for MIMD shared-memory multiprocessors. We parallelized the resampling and compositing steps by distributing scanlines of the intermediate image to the processors. On a 16 processor SGI Challenge multiprocessor the 256x256x223 voxel head data set can be rendered at a sustained rate of 10 frames/sec.

Acknowledgements

We thank Pat Hanrahan, Sandy Napel and North Carolina Memorial Hospital for the data sets, and Maneesh Agrawala, Mark Horowitz, Jason Nieh, Dave Ofelt, and Jaswinder Pal Singh for their help. This research was supported by Software Publishing Corporation, ARPA/ONR under contract N00039-91-C-0138, NSF under contract CCR-9157767 and the sponsoring companies of the Stanford Center for Integrated Systems.

References

[1] Cameron, G. G. and P. E. Undrill. Rendering volumetric medical image data on a SIMD-architecture computer. In *Proceedings of the Third Eurographics Workshop on Rendering*, 135–145, Bristol, UK, May 1992.

[2] Crow, Franklin C. Summed-area tables for texture mapping. Proceedings of SIGGRAPH '84. *Computer Graphics*, 18(3):207–212, July 1984.

[3] Danskin, John and Pat Hanrahan. Fast algorithms for volume ray tracing. In *1992 Workshop on Volume Visualization*, 91–98, Boston, MA, October 1992.

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

[5] Glassner, Andrew S. Multidimensional sum tables. In *Graphics Gems*, 376–381. Academic Press, New York, 1990.

[6] Glassner, Andrew S. Normal coding. In *Graphics Gems*, 257–264. Academic Press, New York, 1990.

[7] Hanrahan, Pat. Three-pass affine transforms for volume rendering. *Computer Graphics*, 24(5):71–77, November 1990.

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

[9] Levoy, Marc. Display of surfaces from volume data. *IEEE Computer Graphics & Applications*, 8(3):29–37, May 1988.

[10] Levoy, Marc. Volume rendering by adaptive refinement. *The Visual Computer*, 6(1):2–7, February 1990.

[11] Levoy, Marc and Ross Whitaker. Gaze-directed volume rendering. *Computer Graphics*, 24(2):217–223, March 1990.

[12] Levoy, Marc. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[13] Meagher, Donald J. Efficient synthetic image generation of arbitrary 3-D objects. In *Proceeding of the IEEE Conference on Pattern Recognition and Image Processing*, 473–478, 1982.

[14] Novins, Kevin L., François X. Sillion, and Donald P. Greenberg. An efficient method for volume rendering using perspective projection. *Computer Graphics*, 24(5):95–102, November 1990.

[15] Porter, Thomas and Tom Duff. Compositing digital images. Proceedings of SIGGRAPH '84. *Computer Graphics*, 18(3):253–259, July 1984.

[16] Sakas, Georgios and Matthias Gerth. Sampling and anti-aliasing of discrete 3-D volume density textures. In *Proceedings of Eurographics '91*, 87–102, Vienna, Austria, September 1991.

[17] Schröder, Peter and Gordon Stoll. Data parallel volume rendering as line drawing. In *Proceedings of the 1992 Workshop on Volume Visualization*, 25–32, Boston, October 1992.

[18] Subramanian, K. R. and Donald S. Fussell. Applying space subdivision techniques to volume rendering. In *Proceedings of Visualization '90*, 150–159, San Francisco, California, October 1990.

[19] Vézina, Guy, Peter A. Fletcher, and Philip K. Robertson. Volume rendering on the MasPar MP-1. In *1992 Workshop on Volume Visualization*, 3–8, Boston, October 1992.

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

[21] Wilhelms, Jane and Allen Van Gelder. Octrees for faster isosurface generation. *Computer Graphics*, 24(5):57–62, November 1990.

[22] Yagel, Roni and Arie Kaufman. Template-based volume viewing. In *Eurographics 92*, C-153–167, Cambridge, UK, September 1992.

[23] Zuiderveld, Karel J., Anton H.J. Koning, and Max A. Viergever. Acceleration of ray-casting using 3D distance transforms. In *Proceedings of Visualization in Biomedical Computing 1992*, 324–335, Chapel Hill, North Carolina, October 1992.