Production Rendering

Ian Stephenson (Ed.)

Production Rendering

Design and Implementation

Springer

Ian Stephenson, DPhil
National Centre for Computer Animation, Bournemouth, UK

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the Library of Congress

Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms of licences issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms should be sent to the publishers.

ISBN 1-85233-821-0 Springer-Verlag London Berlin Heidelberg
Springer Science+Business Media
springeronline.com

© Springer London Limited 2005
Printed in the United States of America

The use of registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant laws and regulations and therefore free for general use.

The publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

Typeset by Gray Publishing, Tunbridge Wells, UK
Printed and bound in the United States of America
34/3830-543210 Printed on acid-free paper SPIN 10951729

We dedicate this book to:

Amanda Rochfort,
Cheryl M. LaMont,
The Elendt Family,
David Moore,
The Iverson Family
The Pantaleoni Family and
Michelle Bickers

Introduction

Developing a production 3D rendering system is an intimidating task. The skills required are both broad and deep, requiring the mastery of practically every aspect of computer graphics from curves and surfaces through to compositing and image file formats. However, it doesn't stop there; the developer needs to be familiar with many aspects of computer science and engineering typically not considered to be graphics related: compiler development, processor architecture, signal processing, virtual machines, and perhaps most importantly of all, the software engineering skills to bring together such a diverse range of techniques into a coherent and manageable body of code.

While many books have been written about rendering, they are typically limited to either the basics of ray tracing or they specialize on a certain aspect of rendering research. When I began developing a renderer of my own, I rapidly found that these texts told me very little about *real* rendering, as it is used in film and video production. Rather than ray tracing spheres and polygons with Phong shading I needed to know how to parse RIB files, and execute compiled shaders, on a range of complex surfaces. I learned about shading engines by examining the compiled shaders from other rendering systems.

As my own rendering system (Angel) developed I found that my experiences were not unique, and I was lucky enough to meet others who were delighted to share their own ideas on dicing strategies, grid sizes, optimization techniques and other minutiae that enable a renderer to tackle the complex scenes found in production. Rendering systems rarely seem to be developed by large teams, but rather by enthusiastic individuals, who are always happy to share and learn from each other. Many renderers which started out as experimental projects, much like mine, have grown into commercial products used across the world.

A few years later when I was invited to contribute a chapter to the *Handbook of Computer Animation* (Vince, 2002) I realized this was an opportunity to share my experience of developing Angel, and present a more realistic introduction to how a production renderer works. While successful, one chapter could only provide an overview of such a huge subject and there was clearly scope for a more in-depth treatment.

Having established that there was a need for this book, rather than attempt to write it myself, I decided to invite the experts I had met to each contribute a chapter on a subject of their choice. I was very pleased when every one of them said yes, and with only a little rearrangement and coercion the rendering pipeline was carved up between us, to produce a comprehensive and in-depth study of how a production render is designed and implemented.

Chapter 1 is an extended and updated version of my original chapter that appeared in the *Handbook of Computer Animation.* The intention is that this provides a roadmap for the later chapters, introducing the concepts and ideas which will be expanded upon later. Though the chapters are presented in a logical sequence they can be read in any order, using the first chapter as a guide.

Chapter 2 is written by Rick LaMont, and deals with the overall structure of the rendering system, showing how the scene is created and represented within the renderer, and how the objects in the scene can be passed through to the various rendering stages. Rick is CTO of Dot C Software, and the lead developer of the RenderDotC renderer.

Chapter 3 deals more specifically with the geometry types typically found in a production render, explaining how surfaces can be evaluated and manipulated. It was written as a team effort by myself, Paul, Scott and Rick.

Having prepared the geometry, it must then be shaded by a procedural shading engine. In Chapter 4 Mark Elendt explains how this can be implemented. Mark was the very first employee of Side Effects Software Inc, developers of the Houdini animation system, where he holds the position of Senior Mathematician. In addition to his contributions to Houdini itself (for which he has received a Technical Achievement Award from the Academy of Motion Picture Arts and Sciences) he is the chief architect of Side Effects' Mantra rendering system, and designer of its VEX shading language.

Matthew Bentham is a programmer at ART VPS, where he develops the shader compiler for their RenderDrive and PURE range of hardware rendering products. In Chapter 5 Matthew describes how a compiler can be written to convert a high level language such as RenderMan SL into a format suitable for use by the shading engine.

While historically most systems capable of handling scenes of the complexity required by commercial production used scanline techniques for efficiency, more recently ray tracing and global illumination have been integrated, to create hybrid renderers. Ray tracing and global illumination are the subjects of Chapters 6 and 7 respectively. In Chapter 6 Scott Iverson considers the problems of adding ray tracing support to a render without sacrificing the performance and flexibility that users expect in a production system. Scott is the founder of SiTex Graphics where he has developed the AIR rendering system, and supporting tools.

Jacopo Pantaleoni, developer of Lightflow Rendering Tools, is the author of Chapter 7, which discusses the problem of global illumination. Jacopo studied mathematics at the University of Padova, has worked at NVIDIA as an intern in the OpenGL group, and is currently working as a Lighting Technical Director for Valiant Productions in London.

Once the surfaces have been shaded they must be assembled into a final image, as described by Paul Gregory in Chapter 8. Paul was the original architect of the Aqsis rendering system and, following its release as an open source project, leads its development team.

Chapter 9, which concludes the book, is a collection of thoughts, information and useful fragments of code contributed by myself, Rick, Scott, Mark and Paul. Though not necessarily fitting neatly within the structure of the book as a whole, we hope you find the contents of this final chapter valuable or at least interesting.

Constructing a high quality renderer requires such a diverse set of skills that this book could never have been written by one person. Each of us has learnt something by reading the others' chapters, and I would like to thank each of the authors for their outstanding contributions.

Ian Stephenson
National Centre for Computer Animation
Bournemouth University

Contents

Contributors

Matthew Bentham
ART VPS Ltd, Cambridge,UK
matthew_bentham@yahoo.com

Mark Elendt
Side Effects Software Inc., Toronto, Ontario, Canada
mark@sidefx.com

Paul Gregory
Aqsis Team, Basingstoke, UK
pgregory@aqsis.com

Scott Iverson
SiTex Graphics, Inc., Denton, Texas, USA
si@sitexgraphics.com

Rick LaMont
Dot C Software, Inc., Kailua, Hawaii, USA
lamont@dotcsw.com

Jacopo Pantaleoni
Valiant Productions, London, UK
jp@lightflowtech.com

Dr Ian Stephenson
Bournemouth University, Poole, UK
istephen@bournemouth.ac.uk

1 A System Overview

1.1 Introduction

Computer animation takes place in a virtual 3D world. However, it is normally visualized through a 2D screen made up of pixels. The rendering process is the computer animator's camera, which records the virtual world in a format that can be broadcast. Programmers have been developing renderers for many years and a range of techniques have been established. However, these approaches with their respective strengths and weakness have traditionally been considered mutually exclusive – a rendering system based on ray tracing would handle shiny reflective surfaces efficiently, but users would simply have to accept that it could not handle displacement or diffuse surfaces as well as some other architectures could.

While computer graphics researchers have the luxury of being able to work with only certain kinds of scenes, and can explore the limitations and strengths of different approaches to rendering, there is increasing demand placed upon rendering software from the commercial sector. In particular, feature film production requires software that can deliver a wide range of images in a timely manner. Driven by this, modern production rendering systems generally make use of a hybrid approach, incorporating a range of techniques to produce software that is both efficient and flexible.

Many of the requirements of commercial production rendering are embodied in the RenderMan standard (Pixar, 2000). While not all rendering software used in production is based upon this standard, it defines a feature set which is typical of a high-end commercial renderer. It is presupposed that most surfaces will be curved rather than polygonal, ensuring that they appear smooth, even when viewed at the high resolution of film. RenderMan also includes procedural shading, which gives users almost total control over the appearance of surfaces. In order to support these features efficiently in limited memory most implementations of the standard sacrificed ray tracing and global illumination support. However, almost all have recently been converted to a hybrid architecture, effectively consisting of several renderers in one package, each solving one part of the rendering problem.

To support such a broad feature set requires a large number of modules each interacting in a carefully controlled manner with its neighbours. This chapter takes a systems approach to rendering by considering the implementation of a simple production style renderer based on the RenderMan standard. An overview of each module is developed, and the flow of data through the system is considered. Following chapters will each embellish upon one area of this roadmap, considering its implementation in greater detail.

1.2 Input

Any rendering system must begin with the parsing of a scene description. The RenderMan standard defines both a C API and a RIB (RenderMan interface bytestream) file format, either of which allows a scene to be fed into the renderer. Any renderer that can support these forms of input may be used interchangeably. A modelling package simply generates a RIB, which allows an animator to select any compliant renderer, based on the requirements of the project. It also allows us to develop a renderer, which may be used for "real" work without concern as to how the scene is to be produced.

A complete description of both the C and RIB APIs can be found in the RenderMan Specification (Pixar, 2000).

1.2.1 RIBs

Although the C API and the RIB interface are functionally almost identical, the RIB interface is simpler, as it avoids the lexical complexities of C. In practice, a renderer would be written to implement the C API internally. A RIB parsing layer can read in RIB files and translate them into C API calls.

A simple lexical analyzer (written using lex) can identify keywords, strings and numbers in a RIB file. For each keyword a unique function is called, which uses lex to extract the parameters to that command. These parameters are then repackaged and a call to the appropriate C API function is made. Because there is a direct correspondence between RIB and the C API, the RIB parser needs to know very little about the semantics of the commands it is processing.

The most complex aspect of the C API and the RIB parsing library, is that RenderMan commands can take a variable number of parameters. While in a RIB file these are simply a list of key/value pairs, passing these into a C function is slightly more complex. RenderMan documentation (Upstill, 1990; Apodaca and Gritz, 1999; Stephenson, 2002) typically demonstrates the handling of these parameter lists through the use of *varargs* functions, where the key/value list is terminated by a NULL. For example, the `Surface` command:

```
Surface "plastic" "Ks" [0.6] "Kd" [0.2]
```

would be implemented in C as:

```
RtFloat spec[1] = {0.6};
RtFloat diff[1] = {0.2};
RiSurface("plastic", "Ks", spec,"Kd", diff, RI_NULL);
```

This form is severely limited, however, as the number of parameters passed to the function must be known at compile time. If the parameters have been parsed from a RIB file at run time then this is not possible.

A second form of the C API must therefore be used in preference. For each function of the form *RiXxxx* which can take a parameter list, there is a second version *RiXxxxV*. In this form, two arrays are passed, one containing the list of tokens, and the second a list of pointers to the values:

```
RtFloat spec[1] = {0.6};
RtFloat diff[1] = {0.2};
RtToken keys[] = {“Ks”, “Kd”};
RtPointer vals[2];
vals[0] = (RtPointer)spec;
vals[1] = (RtPointer)diff;
RiSurfaceV(“plastic”, 2, keys, vals);
```

While superficially more complex when used to demonstrate the RenderMan API, in real applications (such as a RIB parser) the *RiXxxxV* form is far simpler to use, and allows the number of parameters in a parameter list to be changed at run time. A renderer which is to be linked with a RIB parser need only implement the *RiXxxxV* form. A trivial library can convert *RIxxx* calls to the *RiXxxxV* form if required.

1.2.2 Scene Initialization

A RIB file consists of basically two parts: the set-up phase, which commences at the beginning of the file and the world description, which contains the actual scene to be rendered, enclosed in `WorldBegin` and `WorldEnd` commands. Prior to the world description, the set-up phase is essentially describing the virtual camera that will be used, both in terms of its position and its operation.

The nature of the camera is defined by setting a number of *options*. These are parameters which apply to the whole scene, and include the Projection type, Output Resolution, and other standard options along with a mechanism (the `Option` command) to allow renderer-specific extensions. BMRT (Gritz and Hahn, 1996), for example, uses such an extension to control the use of radiosity.

Any transformations prior to `WorldBegin` are taken to define the position of world space relative to the camera.

1.2.3 GState

One of the main tasks of the front end to a renderer is to manage the graphics state (*GState*). This consists of the *current transformation matrix* (CTM), and the *current attributes.*

Transformations are applied in RenderMan such that each transformation applies to all objects, which follow. For example,

```
Sphere 1 -1 1 360
Scale 0.5 0.5 0.5
Sphere 1 -1 1 360
Translate 1 0 0
Sphere 1 -1 1 360
```

creates a sphere of unit radius at the origin; a sphere of radius 0.5 at the origin; and a sphere of radius 0.5 at (0.5, 0, 0). To achieve this, each transformation is converted to a matrix form and multiplied by the CTM which it replaces (newCTM = oldCTM × M).

In order to facilitate hierarchical modelling the current transformation can be stored and restored later using `TransformBegin` and `TransformEnd`. For example, a character's nose can be positioned relative to its head, which in turn is positioned relative to the body:

```
#draw character
TransformBegin
  #position head
  TransformBegin
    #position nose
    #draw nose
  TransformEnd
  #continue drawing head
TransformEnd
#continue drawing body
```

This is easily implemented as a stack of matrices. All operations are performed on the top element of the stack. `TransformBegin` duplicates the top while `TransformEnd` removes it.

Attributes are more general than transforms, of which they are a superset. Other attributes include all properties of the object such as its current surface colour and surface shader. These are managed in a similar way using `AttributeBegin` and `AttributeEnd`. As is the case for options, a renderer may add its own specific attributes using the `Attribute` command (for example, a wire frame renderer might use this to control the number of wires drawn for a particular primitive). Similarly to transformations, attributes apply to all objects that follow unless they are overwritten with a new value or popped from the attribute stack. In most cases the renderer need take no action other than to record the attribute at the point it is actually encountered.

1.2.4 Geometry Commands

While most of the input stages of the renderer are simply concerned with tracking the graphics state, the renderer is required to *do* something upon encountering a command which actually specifies some geometry. Upon encountering such a command, the parameters of the object, along with the current attributes (including the current transformation matrix) will be passed to the next stage of the render.

1.3 Scene Graph

The RenderMan API is defined in such a way that at the point where the command is issued to draw an object, all of the necessary information about that object has already been provided. Using only the scene's options, the current attributes and the object's explicit parameters, the object can be rendered. There is therefore no need to store all of the objects in memory prior to rendering, and this stage may be bypassed completely. As each object is created it can be passed directly to the following stages.

However, if global illumination techniques are being used then accurate shading of the object may be dependent upon all other objects within the scene, including those not created yet. In which case the object must be stored until all objects have been created. Upon the execution of the `WorldEnd` command the renderer can loop over all objects in the scene, and render them with full knowledge of the other objects in the scene.

Even when global illumination is not being used it can still be constructive to render the objects in an order other than that defined in the RIB file. In particular, large objects at the front of the scene may obscure those behind. Objects which are obscured completely need not be rendered – this is known as occlusion culling. The simplest form of scene graph is therefore a linked list of objects sorted by Z-depth. By passing the objects to the following stages, ordered from front to back we maximize the chance of spotting that an object is fully occluded by something which has already been drawn.

When ray tracing is used a more complex data structure is required, which allows objects that potentially intersect with a ray to be rapidly identified. A more detailed examination of the data structures and classes used to represent the objects within a scene, along with the flow of object representations through the rendering pipeline can be found in Chapter 2.

1.4 Objects

Though we have created a data structure which represents the objects in the scene, each individual object is potentially very different. In order to pass the objects through the latter stages of the pipeline we need to convert them to a common format.

The *shading engine* shades a patch in a single pass, rather than shading each point in turn. While certain simple shaders can be evaluated a point at a time, more complex operations require information about the surface, rather than just the point. For this reason the preferred format is a 2D array of points forming a mesh of micropolygons. Provided that each facet of the mesh is smaller than one pixel in the output image, the resulting surface should be a sufficiently accurate approximation to the true surface. This process is known as "dicing".

The required resolution of the mesh to be shaded can be estimated by calculating the bounding box for the patch, and projecting it into screen space. Alternatively, a low-resolution mesh can be generated, and the micropolygons in this mesh can be measured in screen space. By performing the operation at render time, when all information about the camera and object are known, and optimum resolution can be chosen. It also allows the geometry to be stored in a compact high-level format for as long as possible, reducing memory requirements.

By contrast, static tessellation schemes that convert surfaces to polygonal representations during the modelling stage require fine tuning by the user to estimate the tessellation rate required for a particular set of shots. This can easily result in geometry that is too coarse, leading to artefacts, or too dense, which can dramatically increase render times. If an object changes dramatically in size during a shot, due to its movement relative to the camera, there may be no static tessellation which is appropriate, whereas dynamic dicing is automatically recalculated at each frame.

The RenderMan attribute `ShadingRate` allows the user to scale the resolution of the shaded mesh. A finer mesh gives a better approximation to the underlying surface, particularly when displacement is used, at the expense of render time, while a coarser mesh can dramatically speed up test renders.

It is fairly typical that the size of the mesh required to represent a complete object with sufficient accuracy would be very large, and hence would require more memory than is available. In this case the object is simply divided in two – a process known as "splitting". These split patches can then be reconsidered for dicing (and possibly re-split). This also increases the chance that the sub-patch may be off screen or fully occluded by geometry that has already been drawn, allowing the new patch to be more aggressively culled.

For certain objects it may also be found that certain parts of the mesh require very fine dicing while others require coarse dicing (for example, a ground plane extending from the foreground off into the distance). Making the micropolygons too big leads to a lack of detail in the image, while choosing micropolygons that are too small leads to wasted effort and potential aliasing problems. This may also be used as a criterion for splitting the patch, so that each section can be diced at an appropriate level of detail.

1.4.1 Quadrics

RenderMan supports a number of standard quadrics, and other simple mathematical objects such as a sphere, torus, cone, disk, hyperboloid and paraboloid. All of these can be represented simply by a function, which takes u and v surface coordinates and returns a point P in object space. These surfaces can therefore be transformed to a mesh of squares at arbitrary resolution (Figure 1.1).

Given a suitable function to evaluate P, a mesh can be created simply:

```
Point P[uwidth][vwidth]
float du = 1.0/(uwidth-1);
float dv = 1.0/(vwidth-1);
for(u = 0;u<uwidth;u++)
  for(v = 0;v<vwidth;v++)
    P[u][v] = PFunk(surfaceParams, u*du, v*dv);
```

Details on implementing appropriate P functions for quadrics and other surfaces can be found in Chapter 3.

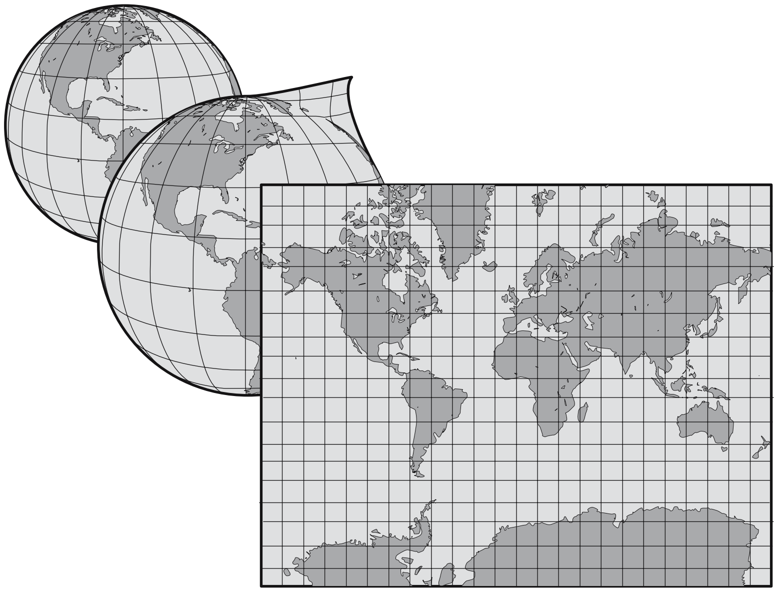

Figure 1.1 A sphere as a mesh.

1.4.2 Patches

Patches within RenderMan may be bilinear, bicubic (with arbitrary basis functions) or rational (NURBS). Patches may be specified individually or as part of meshes, and `PatchMesh` may be periodic or non-periodic. Regardless of these complexities, patches are parametric and hence can be handled identically to quadrics.

In the case of bicubic patches, it may be convenient to convert all patches to a standard basis, for example Bezier, to avoid unnecessary storage of the basis matrices, to optimize the evaluation of the surface, and to make use of the convex hull property of certain basis types.

1.4.3 Polygons

Although polygons and polygon meshes are generally regarded as being trivial to render, they are in fact one of the hardest groups of primitives within the original RenderMan standard for a renderer to handle. They do not convert well to the micropolygon representation, in part because they lack globally consistent surface coordinates. While many texts deal with Phong shading large triangles, we need to reduce polygons to quads or triangles, which can be passed to the shading stage as patches.

Polygons come in three forms: convex, concave and concave with holes. A convex polygon can be trivially converted to quads (one of which may be degenerate, i.e. a triangle), which may then be shaded. Holes may be removed from a polygon by splitting along a line from the outside edge to a point on the hole.

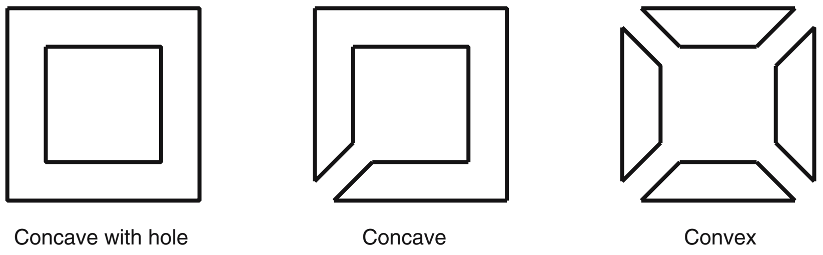

Figure 1.2 Splitting polygons.

Concave polygons may be converted into two or more convex polygons by repeatedly splitting them between two points, such that the new edge is inside the original shape and does not intersect any existing edges. These two processes are shown in Figure 1.2.

1.4.4 Subdivision Surfaces

Being non-parametric, subdivision surfaces inherit some of the problems of polygons, in that they lack well-defined surface coordinates. Repeated subdivision produces a polygonal mesh, which could be used to approximate the surface, but this would be difficult to shade well, and greatly increases memory requirements.

Fortunately, in the case of Catmull–Clark surfaces (Catmull, 1974), following subdivision all faces become quadrilaterals, most of which are simply B-spline surfaces, and hence locally parametric. Those patches near extraordinary vertices (whose valence is other than 4, and hence are not B-splines) can still be evaluated parametrically, as shown in Stam (1998). We can therefore subdivide the original control hull a small number of times, and then treat each of the resulting faces as parametric patches for passing to the shading engine.

1.4.5 Points and Curves

Lightweight primitives are intended to be minimally shaded, either consisting of a single shading sample or a strip of samples (in the case of a curve). We can therefore pack these together into a pseudo-patch for shading. While this will mean that shading calculations based on derivatives are unreliable, such operations have little meaning for these primitives, as they have no width.

1.4.6 Blobby Objects

Pixar's RenderMan (PRMan) 3.9 introduced implicit surfaces (Bloomenthal et al., 1997) in the form of the `Blobby` command (Pixar, 2000). While a number of techniques exist for polygonalizing isosurfaces (Duff, 1992; Watt and Watt, 1992; Heckbert, 1994), simple polygonalization is a poor technique when procedural shading is being used. Standard techniques such as *Marching Cubes* produce a large number of small irregular polygons rather than the regular mesh, which the shading engine prefers, but this is a penalty that must currently be accepted, in exchange for the flexibility of implicit surfaces.

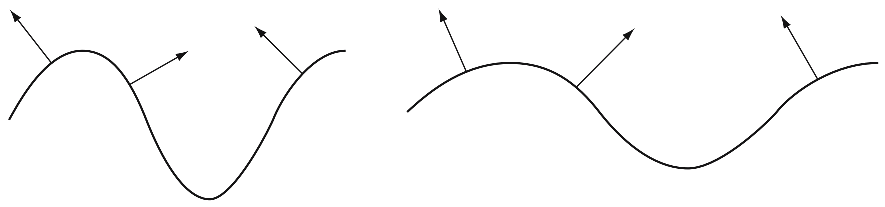

Figure 1.3 Transforming surface normals.

1.5 Transforms and Coordinate Systems

Objects are defined in their own object space. All points, vectors and normals must therefore be converted into a common world space for shading.

The calculations of shading and texturing can be performed in any coordinate system, other than those such as Screen which follow the perspective transform representing the camera lens. When coordinate systems are not explicitly specified within the shader code the renderer references a default space known as *current space*. This space is renderer dependent, but is usually equivalent to camera space. Certain calculations such as ray-object intersection are more easily performed in object space, while world space is in principle the most appealing space in which to perform lighting calculations. In practice, camera space is most commonly used, as it allows all the linear transforms to be performed prior to shading. The first task of the back-end of our renderer is therefore to transform all the micropolygons of our surface into camera space, if this has not already been done.

So far we have considered only points. However, we must distinguish between vectors used to represent points, free vectors and vectors that represent surface normals. These correspond to the shading language (SL) types *point*, *vector* and *normal*, and must be handled differently when transformed both by SL code and by the rendering pipeline. Points respond as expected to scaling, rotation, shear and translations. However, as vectors represent simply an offset to an arbitrary point they are invariant under translation. When a translation matrix is applied to a vector it is unchanged while a point is moved.

The case for normals is somewhat more complex. Consider the 2D case of a sine wave along the x-axis (Figure 1.3). If the graph is scaled in the x-direction (becoming flatter) then the normals become smaller in x (more vertical). Mathematically for a transform M, normals must be scaled by the inverse of the transpose of M. In practice this is often equivalent to M, so the error may go unnoticed for some time – particularly as we rarely care about the magnitude of the normal, merely its direction.

1.6 Shading

Having converted the geometry of the input object into a simple common form – a grid of points we can now consider texturing and lighting each point. Though strictly speaking the term "shading" refers to the calculation of the light interaction

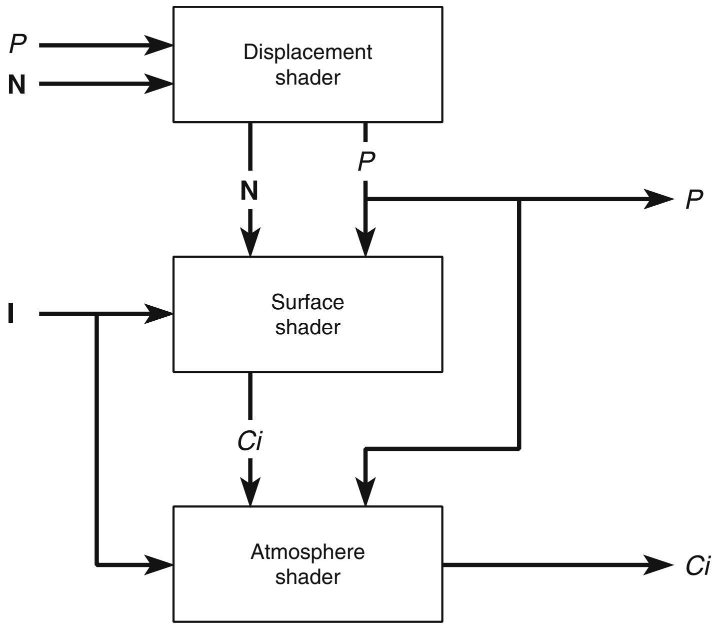

Figure 1.4 The shading process.

with a surface, it is generically applied within production to refer to all operations performed upon the grid.

1.6.1 The Shading Process

RenderMan supports a number of shader types. However, the most important for the shading of surfaces are *Displacement*, *Surface* and *Atmosphere*. The relationship between these is shown in Figure 1.4. Displacement may modify the position and normals of points in the mesh, while Surface calculates the colour of the points, taking into account the observer (viewing along a vector **I**) and the lights in the scene.[1] Atmosphere modifies this colour to simulate the effect of the space between the surface and the observer.

Each of these are written as a function, which are run one after the other. Communication between the shaders is limited to the sharing of global variables. Following shading we will have final versions of *P* and *Ci* (the *position* and *colour* of each micropolygon) ready for drawing.

1.6.2 Shading Language

RenderMan shaders are written using SL – a high-level programming language designed for describing surfaces. Though much like C, extended data types,

[1] In practice certain renderers (particularly PRMan) allow displacement to be applied in surface shaders, though this can give rise to artefacts, and is non-portable.

```
Nf = faceforward (normalize(N),I);                pushv I
                                                  pushv N
                                                  normalize
                                                  faceforward
                                                  popv Nf

V = −normalize(I);                                pushv I
                                                  normalize
                                                  negv
                                                  popv        V

Oi = Os;                                          pushc Os
                                                  popc Oi

Ci = Oi * (Cs * (Ka*ambient() + Kd*diffuse(Nf)) + pushf roughness
     specularcolor * Ks*specular(Nf,V,roughness)); pushv V
                                                  pushv Nf
                                                  specular
                                                  pushf Ks
                                                  pushc specularcolor
                                                  mulcf
                                                  mulcc
                                                  pushv Nf
                                                  diffuse
                                                  pushf Kd
                                                  mulfc
                                                  ambient
                                                  pushf Ka
                                                  mulfc
                                                  addcc
                                                  pushc Cs
                                                  mulcc
                                                  addcc
                                                  pushc Oi
                                                  mulcc
                                                  popc Ci
```

Figure 1.5 Compiling the plastic shade.

operator overloading and special rendering functions allow shaders to be written far more succinctly than in more general programming languages. Conversely, this narrow focus does place some limits to the operations that can be implemented.

SL is renderer independent, so a well-written shader can be used in any compatible renderer. However, it must first be compiled. This translates the shader from the high-level language used by the programmer into a form which can be used directly by a particular renderer. The code for the standard plastic shader in both source and a typical target format are shown in Figure 1.5. The implementation of a compiler to convert between these formats is considered in detail in Chapter 5.

A shader describes shading calculations in terms of a single point, and these are duplicated across the surface, and hence all features must be described implicitly rather than explicitly. For example, to draw a disk we must consider whether each point is inside the disk and colour it appropriately rather than drawing the disk onto the surface. Rather than shading each point to completion, each operation in the code is executed for the whole surface before moving to the next operation.

This approach of defining a set of operations, and then performing these operations in parallel across a large data set is known as SIMD computing (single-instruction multiple data; Flyn, 1972). In addition to fitting well within the shading paradigm (where a surface must be coloured), it can be implemented very efficiently, is ideal for hardware acceleration (Olano, 2000; Stephenson, 2003) and allows operations such as the calculation of surface normals to be implemented far more easily than could be done if each point were shaded in turn.

1.6.3 The Shading Engine

The basic data structure on which the shader engine operates is a 2D array of points, each point being logically connected to its neighbours to form a regular patch. By this stage all geometry has been converted into this format. For each point we need to store shading variables such as *P* (position) and *N* (surface normal). Hence, we have an array of memory addresses associated with each point (or in SIMD terminology, node) in which these variables can be stored.

The SIMD Virtual Machine

Shaders are typically compiled to a virtual machine code, which is interpreted at run time. This simplifies the compilation process and allows compiled shaders to be used across a range of available hardware (in a render-farm, for example).

While this approach may appear to be slow (as it is in the case of the Java virtual machine (VM) or the BCPL INTCODE system (Richards and Whitby-Stevens, 1979) this is in fact not the case when the technique is used for procedural shading. Any overhead in the interpretation of code is incurred due to the instruction decode – an instruction is read in, and the interpreter must decide how it should be executed. If it takes nine CPU cycles to work out that the instruction is an add operation, and then one cycle to perform the add, such a system would indeed be extremely slow. However, in the case of shading, having decoded the add instruction we must apply it to (say) a 100×100 grid of micropolygons, taking approximately 10,000 cycles. The instruction overhead has dropped from 90% of CPU time to 0.01%.

In addition, many shading operations can be added to the instruction set. So, for example, a noise function would typically be supported as a single instruction. This will take a significant time to execute (noise accounts for up to 50% of the CPU time used by many typical shaders), again reducing the significance of the instruction decode. The plastic shader in Figure 1.5 makes use of faceforward, normalize, ambient, specular and diffuse instructions all of which are highly specialized operations not typically part of a regular machine's instruction set. Their presence in the virtual machine greatly simplifies and optimizes the interpreted code.

The code of our shading engine is therefore simply a loop, which reads an instruction from the shader (often simply a text file), identifies the instruction and then performs that operation on all of the points on the surface. In its most basic form:

```
while(fscanf(opstream, “%s”, operation))
{
...
if(strcmp(operation, “add”) == 0)
  {
  fscanf(opstream, “%s”, source1Name);
  fscanf(opstream, “%s”, source2Name);
  fscanf(opstream, “%s”, destName);
  s1 = lookup(source1Name);
  s2 = lookup(source2Name);
  dest = lookup(destName);
  for(i = 0;i<nodeCount;i++)
    {
    machine[i][dest] = machine[i][s1] +
    machine[i][s2];
    }
  }
...
}
```

Derivatives

Within the shading language a number of operations (known as derivative or area functions) depend not just on the point being shaded but on the way variables change across the surface. The most obvious of these is `calculatenormal`, which finds the normal of a new displaced surface. Though at first sight challenging to implement, once a SIMD approach is adopted, providing a numerical approximation to the new surface normal becomes trivial:

```
if(strcmp(operation, “calculatenormal”) == 0)
  {
  for(i = 0;i<maxnodes;i++)
    {
    dPdu = (machine[i+1][pAddr] - machine[i][pAddr])
       /(machine[i+1][uAddr] - machine[i][uAddr]);
    dPdv = (machine[i+uwidth][pAddr] - machine[i][pAddr])
       /(machine[i+uwidth][vAddr] - machine[i][vAddr]);
    machine[i][nAddr] = crossProduct(dPdu, dPdv);
    }
  }
```

Note that this code is slightly simplified as it fails to take into account the edges of the grid. Great care must be taken throughout the renderer to ensure that grid edges are handled correctly.

Table 1 Conditional execution

SL	VM code	Node1	Node2
if (s>0.5)	gt s 0.5	true	false
	jmpIfFalse 1		sleep on 1
{do this}	....	active	sleep
else	jmp 2	sleep on 2	sleep
	label 1	sleep	wakeup
{do that}	....	sleep	active
done	label 2	wakeup	active

Branches and Loops

So far we have applied each SIMD instruction to every node of the virtual machine. In order to implement the shading language we need to implement conditionals (*if–then–else*) and loops (*for*, *while* and *illuminance*).

When a conditional statement is encountered in a SIMD context it is generally not a branch, in that both paths of the code must be executed, each path being applied to a different set of nodes. This is managed by an array of flags indicating which nodes are currently active. When an *if* statement is encountered, all nodes which fail the test are turned off, and the code in the body of the conditional is executed. Every instruction is conditional, in that it is only executed upon nodes which are enabled, as in Table 1.

At the end of the conditional we need to wake up the sleeping nodes. This can be achieved by marking the nodes with the value of a label to which they have "branched". When a label instruction is encountered, any nodes sleeping on that label are woken. Note that no branch has actually taken place – the interpreter simply keeps reading instructions. In the case of an *else* clause, all active nodes are suspended, and a label wakes up all nodes, which should execute the else. Finally, at the end of the conditional all the nodes are woken by a further label.

In addition to conditional forward branching, we simply need unconditional backward branches to be able to implement all SL looping constructs. For example, a *while* loop can be coded as:

```
label 1
  evaluate condition
jumpIfFalse 2
  body of loop
jmp 1
label 2
```

Backward branches are implemented by changing the program counter, which indicates which instruction the interpreter will process next – we actually do need to branch. Though backward branches are unconditional within the assembly language the shading engine only needs to execute the branch if there is an active node which requires it. If no nodes are active then the loop is complete, and the interpreter can be allowed to progress naturally to the next instruction.

Lighting

So far we have ignored the subject of lighting. The reason for this is that in the case of local illumination, lighting is simply a shading calculation – lights encountered in the RIB stream are simply recorded and passed to the shading engine. Within the virtual machine we have incorporated instructions such as diffuse and specular, which take their parameters of position, normal, roughness and observer from the stack, and loop over all lights within the scene calculating the lighting using standard lighting equations (Hall, 1989; Cook and Torrance, 1982).

If the RenderMan diffuse and specular functions were coded directly in SL they would be:

```
colour Cdiff = 0;
colour Cspec = 0;

illuminance(P, Nf, PI/2)
  {
  Cdiff+ = Cl*normalize(L).Nf;
  Cspec+ = Cl*pow(normalize(V+normalize(L)).Nf,
      1/roughness);
  }
```

While such an implementation is informative, it is significantly less efficient than the built-in implementations.

If non-standard light shaders are applied these must also be evaluated using the shading VM. This can either use a separate instance of the machine, or the shading calculations may be performed on the same array as the nodes, which actually require the data. In either case the results of these calculations should be cached, as the result of the light shader will be used by both the diffuse and specular calculations.

Texture Maps

Though procedural control of surfaces is an essential feature of a high-end renderer, it is often necessary to use painted textures to add specific detail or provide artistic control over procedural components. However, when multiple high-resolution maps per shader are applied to hundreds of objects, the memory requirements can become significant.

To avoid loading large maps for objects which are small on screen, texture files are generally mip-mapped. The image is repeatedly halved in both width and height to generate multiple representations of the high-resolution texture which are a fraction of the original size. For many of the objects in the scene, only the low-resolution versions are required in memory, and the full-sized images may be left on disk. As the low-resolution images are generated prior to rendering, they also save CPU time, as filtering is performed only once per texture, rather than at each access.

When objects are viewed closely, at levels of detail where the most detailed textures are required, it is unlikely that the whole texture will be visible on screen in a single frame. Each representation of the image within the mip-map is broken into tiles, and only those tiles which are on screen need be loaded into memory.

Because shading takes place on grids, there is strong locality of reference – a texture which is accessed by one point will be required for the entire grid, at a similar level of detail. Tiles are loaded into memory, as they are required, and held for use across the grid, and on neighbouring grids using similar shaders. Tiles which have not been accessed for some time are discarded to make room for new tiles. By effective caching, tiling and mip-mapping, textures can be squeezed into a fraction of the memory otherwise required.

Uniform versus Varying Data

Not all calculations need be repeated for every point. Many values are uniform across the shaded surface, and hence SL provides a mechanism for identifying this, allowing repeated calculations to be avoided.

The most obvious strategy for implementing such an optimization would be to add qualifiers to each instruction specifying whether it operates in uniform or varying data. The compiler then tracks this information through the parse tree and generates code that uses the new instructions. At run time operations such as `addUU` need only perform a single addition. Instructions such as `addUV` may also be optimized compared with `addVV`.

A more interesting approach is to monitor the type of expressions at run time. A push operation can record whether the value at the top of the stack value is now uniform or varying. An add operation can check the types of data on which it is being asked to operate and optimize accordingly. This and other advanced shading techniques are the subject of Chapter 4.

1.7 Ray Tracing

Historically, ray tracing has been considered as a complete solution to the rendering problem. However, such systems are far slower than micropolygon style renders, and hence have seen limited application in production environments. Users were forced to choose between the performance of a scanline system, and the ability to easily handle complex shadows, reflections and refractions afforded by ray tracers. Typically, production deadlines, high-resolution requirements and the complexity of scenes forced ray tracing to the sidelines.

More recently this choice has been negated. Ray-tracing systems have added micropolygon-based subsystems to handle the rays from the camera, while micropolygon renderers have added ray-tracing engines to handle reflections from objects. Within this model ray tracing is simply a shading operation – at its most simple we need to add a `trace()` function to the shading language which returns the colour of the scene in a particular direction, providing ray-tracing flexibility "on demand". If the user chooses not to call this-function they achieve the full performance of the scanline renderer, and can introduce ray-traced lighting on a per primitive basis, as time and image quality demands.

A second function, `transmission()`, calculates whether there is an object blocking a ray between two points. In this case we are not interested in the colour of the object, but simply its opacity. We are also not interested in the *first* intersection, but the combination of all objects along the ray. This function can be built into light shaders to create ray-traced shadows.

1.7.1 Ray–Object Intersection

In order to calculate the colour which the trace function should return we need to find the first object which would be intersected by a ray from the point being shaded, in the specified direction. Most basically we need a function for each type of object which tests if the ray hits an instance of that object, and returns the u,v coordinates of the intersection. For certain simple objects, such as spheres, this can be done easily. However, for more complex objects, such as NURBS, this calculation becomes less tractable. This approach also makes displacement difficult, as this can deform the surface in arbitrary and unpredictable ways. For this reason many renderers do not support displacement of secondary surfaces.

An alternative to calculating intersections directly is to tessellate the object into a polygonal approximation. While this makes finding the intersection point simple it creates the problem of storing large amounts of tessellated geometry. In addition, it can be difficult to select an appropriate level of detail that is fine enough for all rays without being too dense. A solution to this is to dynamically tessellate objects on demand, when they potentially intersect a ray. These tessellations are stored in a cache, and can be reused for similar rays. Multiple versions of the same object may even exist, if different levels of detail are required. Tessellations which have not been used recently can be discarded to free up memory.

Clearly, we would like to avoid performing ray–object intersection tests (either analytical or numerical) wherever possible, and the first stage in achieving this would be to test the ray against a bounding box for the object. This test can be performed, and if the ray either misses the bounding box, or intersects it at a distance further than the closest intersection found so far, it can be discarded.

Rather than exhaustively testing each object we can group objects together, potentially allowing several objects to be discarded with a single bounding box test. The objects in Figure 1.6 have been divided into three groups, based on the central dividing line – those objects on the left, those on the right, and those which cross the boundary and spill into both halves.

To intersect a ray with these objects we first intersect the ray with each of the three bounding boxes. If an intersection is found in the nearest group then we may not need to consider the latter boxes. In the case of Figure 1.6, only objects in the right and middle boxes would need to be considered. Once the intersection is found with the inverted triangle, we know that all objects in the left bounding box must be further away, and can therefore be ignored. By extending this approach recursively large scenes can be efficiently searched.

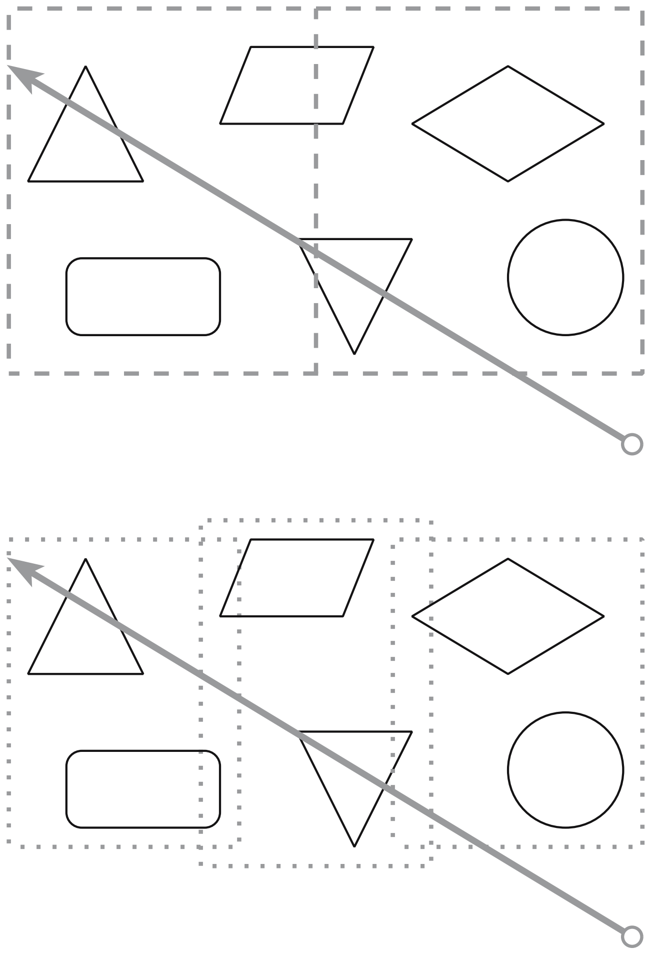

Figure 1.6 Grouping objects.

1.7.2 Shading Ray-traced Objects

Having established that a ray intersects an object at a particular point, we need to calculate the colour of the object at that point. In principle this can be achieved trivially by simply passing that point back into the standard shading engine. However, there are a number of issues which need to be considered.

First, the shading engine must be designed to be re-entrant. It is being called recursively, and hence care must be taken when accessing global data structures. In particular, accessing attributes must access the attributes of the object currently been shaded, and not the attributes of the object which launched the ray.

A more significant issue is that the shading engine is designed to shade grids rather than individual points. To handle derivatives correctly we must construct a small grid of four points to support the point we are actually interested in shading. In addition to the intersection point u, v the points $u + du, v$, $u, v + dv$ and $u + du, v + dv$ will also be shaded.

When shading primary objects the grids used can be relatively large, which allows the overheads of interpretation to be shared by many micropolygons. When shading secondary objects the grid size is fixed at four points, which can dramatically reduce the efficiency. Operations performed once per grid can become significant, and must be carefully optimized if good performance is to be achieved.

Attention must also be paid to avoiding recursive explosion of traced rays. Though we are shading four points we are only interested in the result from one. If each of these were to launch secondary rays, either by explicitly tracing, or by evaluating lighting which creates shadow rays then 16 further points would be shaded. As three of the four points are only required for the calculation of derivatives, they can be culled once the final area calculation within the shader has been performed (indicated by a hint from the compiler). This can dramatically improve performance. Similarly, when calculating shadow rays the shader can be terminated as soon as the opacity of the surface has been derived, usually avoiding performing lighting calculations, which may again spawn further shadow rays.

Chapter 6 provides more information on the problems of implementing a ray-tracing sub-system within a hybrid production renderer.

1.8 Global Illumination

Ray tracing allows for basic shadows and reflections in shiny surfaces. While this can dramatically improve the appearance of artificially constructed scenes, in the real world most surfaces do not behave as perfect reflectors. In Figure 1.7a (Plate 1.1) a simple scene is lit by only direct illumination. The same scene is augmented using ray tracing in Figure 1.7b (Plate 1.2), enabling the boxes to interreflect. The boxes also cast ray-traced shadows onto the floor and walls, greatly increasing the realism of the scene.

However, while Figure 1.7b appears a vast improvement on Figure 1.7a, it is clearly lacking in verisimilitude when compared to Figure 1.7c (Plate 1.3), which is rendered using global illumination. Most obviously the ceiling in Figure 1.7b

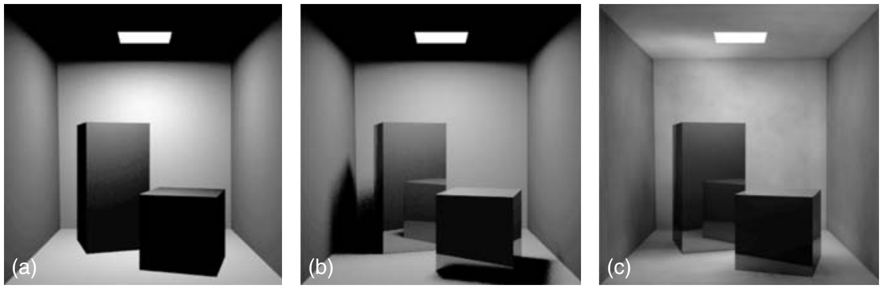

Figure 1.7 Ray tracing and global illumination.