RenderMan, Theory and Practice

Siggraph 2003 Course 9

July 27, 2003

Course Organizer: Dana Batali, Pixar

Lecturers:

Byron Bashforth, Pixar
Dana Batali, PIxar
Chris Bernardi, Pixar
Per Christensen, Pixar
David Laur, Pixar
Christophe Hery, Industrial Light + Magic
Guido Quaroni, Pixar
Erin Tomson, Pixar
Thomas Jordan, Pixar
Wayne Wooten, Pixar

About This Course

RenderMan developers present many of the new RenderMan features and explore potential applications in CGI production. RenderMan practitioners present the latest tricks and techniques used at production studios to create stunning visual effects and feature-length films.

Prerequisites

This course is for graphics programmers and technical directors. Thorough knowledge of 3D image synthesis, computer graphics illumination models, the RenderMan Interface and Shading Language and C programming are assumed.

Topics

RenderMan enhancements, including ray tracing, occlusion, and global illumination.

Practical solutions (with examples) to complex rendering problems:
- skin rendering and subsurface scattering
- translucent objects
- reflections and refractions

Implementing and enhancing RenderMan based production pipelines

Details about specific rendering challenges in the making of "*Finding Nemo*"

Suggested Reading

The RenderMan Interface Specification, versions 3.2 and 3.3 (draft)
`https://renderman.pixar.com/products/rispec/index.htm`

The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics, Steve Upstill, Addison Wesley, 1990.

Advanced RenderMan: Creating CGI for Motion Pictures, Tony Apodaca and Larry Gritz, Morgan-Kaufman 1999.

Realistic Image Synthesis using Photon Mapping, Henrik Wann Jensen, A.K. Peters, 2001.

Texturing and Modeling, A Procedural Approach Second Edition, Ebert, Musgrave, Peachey, Perlin, Worley, Academic Press Professional, 1998.

Lecturers

Byron Bashforth received his Bachelors and Masters in Computer Science from the University of Saskatchewan in 1996 and 1998. After joining Pixar Animation Studios in 1999 as a technical director, he has worked on “Toy Story 2”, ”Monsters. Inc”, “For the Birds”, “Finding Nemo”, and “The Incredibles”. His recent work has focused on character shading and look development.

Dana Batali is Director of RenderMan Product Development at Pixar Animation Studios. Dana has worked at Pixar for 15 years and has been involved with RenderMan since its inception. Prior to leading RenderMan development, his focus was to develop RenderMan applications. In this role, Dana developed Showplace, Typestry, Slim, ATOR and MTOR and managed the development of the RenderMan Artist Tools. Dana holds a BSE in EECS from Princeton University.

Chris Bernardi is a Shading Technical Director at Pixar Animation Studios. He joined Pixar in 2000 during the preproduction phase of “Finding Nemo”. Chris has been involved in Computer Graphics for over 15 years as both a software developer and a 3D artist. Prior to working in Computer Graphics, Chris worked in the music industry as asound designer, composer, touring musician, and educator. He received his Bachelor’s Degree in Biology from Washington University in 1985.

Per Christensen is a senior developer at Pixar Animation Studios. He is working on ray tracing and global illumination for Pixar’s PhotoRealistic RenderMan renderer. Prior to joining Pixar, he was a senior software engineer at Square USA in Honolulu, developing and implementing efficient global illumination methods in the massively parallel renderer Kilauea. He has also co-implemented the first commercial implementation of the photon map method. Per received his Ph.D. from the University of Washington for research in hierarchical techniques for glossy global illumination.

David Laur is a senior developer in Pixar’s RenderMan group, primarily working on the design and implementation of various ray tracing features in the renderer. He has also been the primary developer for Pixar’s distributed computing manager, Alfred. Prior to his six years at Pixar, David managed the Computer Science graphics lab at Princeton University where he worked on scientific visualization projects, including a 1991 SIGGRAPH paper on interactive volume rendering.

Christophe Hery is an Associate Visual Effects Supervisor at Industrial Light + Magic (ILM). Christophe joined ILM in 1993 as a senior technical director on The Flintstones. Prior to that, he worked for AAA in France setting up a CG facility devoted to feature film production. Christophe majored in architecture and electrical engineering at ESTP, where he received his degree in 1989. While a student in Paris, he freelanced as a technical director at BUF and TDI, where he worked on 3-D software development in his spare time. After graduation, Christophe took a job as director of research and development at Label 35, a Parisian cartoon studio. AT ILM, Christophe is currently working in ILM’s R&D department and his numerous projects at ILM include work on: “Harry Potter 2”, “Jurassic Park 3”, “Star Wars: Episodes 1 & 2”, “Mission to Mars”, “Jumanji”, “Casper”, and “The Mask”.

Guido Quaroni has worked at Pixar since 1997. At the studio he worked on Toy Story 2 in the modeling, shading and FX departments. He also worked on “Monsters, Inc”. as sequence supervisor (and the voice of “Tony the Grocer”). At thistime, he is working in the studio tools department focusing on Pixar’s shading pipeline.

Erin Tomson received a Bachelors in Computer Engineering from the University of Michigan in 2001 where she studiedcomputer graphics, computer vision, and data analysis. Since graduating she’s worked as a Technical Director at Pixar Animation Studios. In addition to occasional software development, she worked on “Finding Nemo” doing character shading and designing a subsurface scattering system for skin.

Thomas Jordan started out at Pixar Animation Studios in 1997 with a summer internship in the Story Department on “A Bug’sLife”. After receiving his BS in Electrical Engineering at UC Berkeley in 1998, he came back to Pixar, but this time as a Technical Director. He worked in the Rendering Group on “A Bug’s Life” & “Toy Story 2”, and then joined the ShadingTeam on “Monster’s Inc”. Thomas is currently Shading and Lighting on “Finding Nemo”.

Wayne Wooten has worked as a software engineer at Pixar Animation Studios since 1997. He worked as a lighting technical director on “A Bug’s Life” and led a team that re-designed the render farm control software for “Toy Story2” and “Monsters Inc”. Currently Wayne is working in the Studio Tools department at Pixar focusing on new features for Pixar’s RenderMan. Most recently he helped optimize RenderMan and worked on translucency effects for “Finding Nemo”. Wayne received his BS and Ph.D. degrees from the Georgia Institute of Technology.

Contents

Preface

Welcome to the RenderMan course at SIGGRAPH 2003. This year we've titled our course: *RenderMan, Theory and Practice*, in an attempt to emphasize the incredible relationship that exists between the communities of RenderMan developers and RenderMan practitioners. As developers, we focus on the latest computer graphics theories and algorithms. As practitioners, we try to bend RenderMan software and systems to produce pictures that are theoretically impossible. There's no greater satisfaction for developers than to see our creations used in ways never imagined and we're keen to absorb, formalize and optimize the experimental techniques developed by practitioners. The circle is complete when practioners rush to explore the possibilities offered by new features and optimizations. It's this synergy between developers and practitioners that propels our industry forward and it's the common language of the RenderMan Interface that facilitates communication across boundaries.

This is the tenth SIGGRAPH course on the use and evolution of RenderMan. As the RenderMan Interface approaches its fifteenth birthday, all signs are positive that this to-and-fro between theory and practice will continue to produce improvements in rendering systems as well as ever-more-amazing imagery.

We hope you enjoy this year's material and that you continue to push the boundaries in RenderMan theory and in practice.

Dana Batali, Pixar
April, 2003

Chapter 1

The Evolution of RIB - **Plumbing the Production Pipeline**

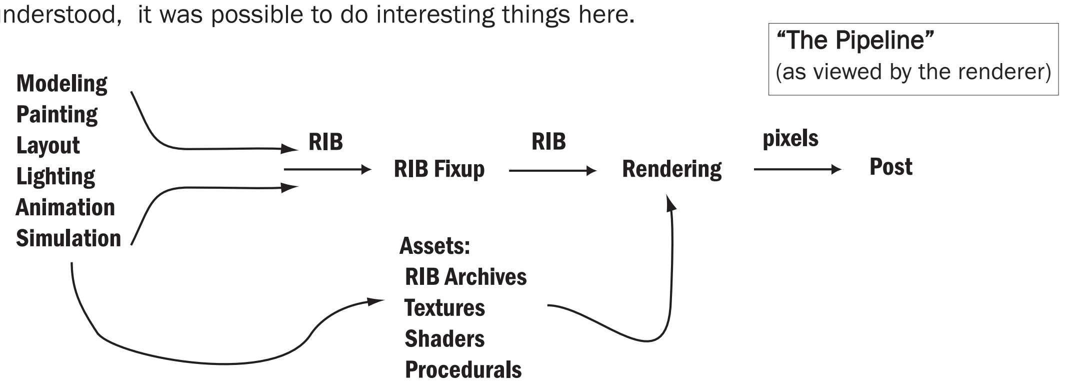
Dana Batali, Pixar Animation Studios
dana@pixar.com

The society which scorns excellence in plumbing because plumbing is a humble activity, and tolerates shoddiness in philosophy because philosophy is an exalted activity, will have neither good plumbing nor good philosophy. Neither its pipes nor its theories will hold water. John W Gardner

Introduction

The RenderMan Interface was formally unveiled in May of 1988. The first release focused on the problem of describing a scene to a photorealistic rendering system and was described in the form of a C-Language API between "modelers" and renderers. The RIB protocol was unveiled in 1989 and addressed the need to de-couple the modeling and rendering systems. RIB allowed the modeling system to spool jobs to remote network renderers or to stash this temporary representation for later consumption by a renderer. To avoid opening Pandora's box, there was a conscious decision to limit the capabilities of RIB. A number of graphics "standards" of the 1980s were in the process of failing at their attempts to impose standardization onto the representation of 3-D models. Pixar didn't wish to enter this fray and steered clear of RIB features not strictly adhering to the modest design goals. RIB quickly became the predominant conduit to RenderMan but the apprehensions of RenderMan engineers soon materialized. Requests for enhancements to the new "file format" came pouring in: Can you add animation data? How about loops and variables? And what about C-preprocessing? Generally these requests were categorized as either "new language binding" or as misguided and RIB remained relatively stable for nearly seven years.

Then something happened. Perhaps it was the end of the cold war? Perhaps it was the new economy? Or perhaps the defenders of the RI Spec had just been worn out by constant badgering by customers? In any event, starting in 1996 RIB began to change in subtle but fundamental ways and it continues to do so today. Driving these changes are requests made from the trenches by the foot soldiers of CGI production - the Technical Directors. TDs often find themselves stuck between a rock and a hard place and with little time to work out an escape plan. They deal with a variety of tools which flow data through modeling, layout, animation, simulation, shading and lighting to rendering. The glue that holds these often disparate tools together is loosely called "The Pipeline" and it can be difficult to understand the inner workings of any given pipeline. Production pipelines differ from studio to studio, from production to production and from shot to shot, but they seem to share one property universally: pipelines are hard to change. Pipelines built to support the RenderMan Interface usually produce RIB for consumption by the renderer and the sleep-deprived TDs were quick to realize that here was a place in the pipeline they could get their fingers on. Because the RenderMan Interface is well designed, minimal, and easily understood, it was possible to do interesting things here.



TDs have long considered various home-grown RIB processing utilities to be critical to their work so it's not surprising that occasional feature requests for the RIB format might arise. And then there are the beleaguered pipeline developers. The unquenchable artistic thirst for complexity coupled with the incredible increase in processor performance continually expose limitations of a given pipeline design. At some point during the 1990's we crossed a milestone at which the time to generate RIB files became significant. Pipeline developers had to re-architect their pipelines to address these new bottlenecks and this resulted in a number of impassioned requests for RIB enhancements.

In retrospect, it appears that the worst fears of RIB minimalists haven't materialized. People have accepted the idea that RIB is not a general 3-D file format. They accept that RIB is merely a conduit leading to their renderer and defend their RIB enhancement requests in practical terms grounded in the need for greater encapsulation, flexibility and more optimal plumbing.

What follows is a guided tour through the evolution of RIB and its uses in the production pipeline. The hope is that a unified presentation of this collection of enhancements will help pipeline designers identify areas for improving their current pipelines. TDs will need to be aware of these changes as well insofar as they affect the design and operation of their RIB fixup tools.

The First Steps

RIB, RIB Structuring (1989)

With the arrival of RIB on the RenderMan scene came an often overlooked appendix to the RenderMan Interface Specification, version 3.1. Appendix D followed the lead of Adobe's Encapsulated Postscript (EPS) guidelines and suggested that RIB writers take care to embed information into their RIBs not strictly required by a renderer. The idea was that by embedding a standard set of RIB comments as well as following a few structuring conventions RIB writers could make their output interpretable by a collection of tools loosely described as Render Managers. Just as EPS files played a fundamental role in the success of desktop publishing, it was envisioned that compliant RIBs would facilitate the adoption of RenderMan as a 3-D "printing" standard and this would quicken the adoption of 3-D photorealistic rendering by the masses. For any number of reasons, the RIB conventions and 3-D rendering in general weren't widely adopted and Appendix D has received no attention (and no updates) since original publication.

RiArchiveRecord, RiReadArchive (1996)

Fast-forward seven years. Until 1996 the RenderMan Interface had no mechanism for embedding RIB structuring conventions into the RIB stream. RIB writers had to "roll their own" RIB libraries or use undocumented features of Pixar's RI client library to embed comments into the RI stream. Surprisingly, RI made no explicit acknowledgement of RIB (RI Archive Files) until the addition of the Archive calls in 1996. With `RiArchiveRecord` you could finally express RIB comments using RI. More importantly, you could now describe the composition of RIB streams to the renderer via `RiReadArchive`. TDs had long resorted to various RIB stream processing tricks (`cpp`, `sed`, `cat`, ...) to accomplish this fundamental operation. With the addition of `RiReadArchive`, RIB had finally become a first class citizen in the RenderMan world. The inclusion of `RiReadArchive` was an acknowledgement of the need for efficiencies in a RenderMan-based pipeline. RIB files had grown in size and in the costs to create and transfer them. Multipass rendering had matured and it was wasteful to duplicate the same complicated models in the RIB for each pass. Instead pipeline designers now could freeze static portions of a shot into individual RIB files then include these from driver RIB files. In this way, a single frame with seven shadow passes and a reflection pass could be reduced from 9 large RIBs to one or more large archive files and 9 small driver RIBs. `RiReadArchive` led to significant savings in RIB generation and network transfer times as well as in disk space. Not surprisingly, it was adopted quickly and widely. It's worth noting that in-renderer support for RIB composition allowed pipeline designers to eliminate ASCII RIB preprocessing and this, combined with the adoption and efficacy of gzip compression, allowed pipelines to produce and transmit compressed binary RIB resulting in significant additional savings.

Plug-in Procedural Primitives and Delayed Read Archive (1997)

With the widespread adoption of RIB archives and the ongoing increases in processing power came the need for additional optimizations. Pixar's "A Bugs Life" represented a quantum leap in the complexity of CG imagery. Gigabyte RIBs were becoming commonplace and it became clear that enhancements to RIB processing were required to more efficiently handle scene descriptions of this size. With additional hints, it would be possible for a renderer to delay the processing of a RIB archive until the moment the encapsulated data was required. Then, rather than process thousands of high-resolution ant models before rendering a single pixel, it should be possible to process only those models needed to produce a small subregion of pixels. Advances in standards for plug-ins (shared objects, dynamic link libraries) allowed RenderMan designers to support "Delayed Read Archive" (DRA) functionality in the form of a "Procedural Primitive". `RiProcedural` had been around "since the beginning" and addressed the problems of data amplification raised by fractal geometry. The adoption of RIB as the standard conduit to the renderer made Procedurals inconvenient and mundane difficulties associated with direct-linking of 3rd party procedures with a renderer didn't help. Now that a standard means for representing and constructing plug-ins was available, the time was ripe to dust-off procedural primitives. The "late binding" properties of renderer plug-ins allow us to think of procedural primitives more like shaders and textures: as extensions to a renderer rather than as built-in characteristics of it. The "just-in-time" aspect of procedural primitives was exactly what was required to enable DRAs. Three new built-in procedural primitives (`RunProgram`, `DelayedReadArchive`, and `DynamicLoad`) were added to RIB at this time and virtually eliminated the concerns about RIB's ability to cope with the growing complexity of CG imagery. For example, the epic battle scenes in Lucasfilm's "The Phantom Menace" might not have been possible without ILM's application of these important capabilities.

Inline Parameter Declarations (1998)

As RIB archives grew in importance a number of long standing minor annoyances with RIB became more annoying. The RenderMan Interface has always been limited in terms of symbol declarations. To introduce new symbols to the interface, a global, unscoped symbol table is modified via calls to `RiDeclare`. With a growing number of RIB archives, came a lack of clarity with respect to the state of the symbol table. Without any scoping constructs, RIB writers had to be defensive and literally declare all symbols prior to use. This introduced unsightly bloating and required RIB Fixup utilities to carefully track local symbol table modifications. Inline declarations were introduced to simplify these problems. Now the declaration could be made at the point of use and, when applied ubiquitously, the need to track symbol definitions across multiple RIB requests is eliminated. Following the long-standing tradition to minimize complexity, RenderMan Interface designers decided that the addition of inline declarations with no side-effects was the best solution to this problem. And, indeed, the low-level grumbling emanating from RIB creators, consumers and processors on the topic of symbol tables has clearly abated. Now the focus shifted to a number of similar issues.

Smart Archives (2001)

For RIB archiving to be truly sound, individual RIB archives must be both flexible and stable: flexible insofar as they can be used for different rendering passes and achieve "appropriately different" results; and stable such that when instantiated into a RIB they produce predictable, correct results. The term smart archives refers to a collection of recent RIB features focused on resolving RIB archive issues.

String Handles

The RenderMan Interface uses opaque handles to describe the identity of lights and retained geometry. Prior to 2001, RIB used short integer values to represent light and object handles and this posed serious problems for RIB archives. Collisions occur when two independent archives contain lights or objects with the same handle and this made it practically impossible to archive lights and objects. The solution to this problem was to support arbitrary string handles in RIB and to generate globally unique identifiers to automate collision avoidance. Mechanisms to override the automatic unique-id scheme were added to the Pixar's RIB client library in order to allow RIB writers to provide their own site-specific scheme. If lights are created with unique names in the upstream modeling environment it is simple and appropriate to use those same names as the RIB light handle.

Scoped Coordinate Systems

In the RenderMan Interface, coordinate system definitions are similar to symbol declarations in that they reside in a single, global namespace. Unlike symbol definitions it is not possible to declare and use the value simultaneously. In RenderMan, coordinate systems must first be defined in RIB. Next they are referred to at a different point in the RIB or from within shaders during rendering. In the pre-archive days, RIB writers would construct unique coordinate system names and then fastidiously pass these down to the correct shader instances. Scoped coordinate systems were added to alleviate the burdens imposed by the single global namespace. First, the fact that they obey standard RenderMan attribute scoping rules eliminates the need to guarantee uniqueness across all archives. Second, to the extent that the coordinate systems represent a canonical shading space it is no longer necessary to pass the coordinate system name to the shader. If a system of shaders requires a shader space for applying paint, it was now possible to "hardcode" the name "paint" into the shader and simply require that models provide some definition of "paint" space thereby reducing per-shader and per-model maintenance costs at large facilities.

Shutter Offset

Another minor difficulty for archives arises when archived objects request motion blur. Since RenderMan motion blocks are described in terms of the global shutter interval and since it is useful to "recycle" archives throughout a sequence, it is useful to follow the convention that the shutter be represented in a frame-relative time frame. This means that the shutter shouldn't embody current time information. So typical shutter settings for two frames of `[10 10.5]` and `[11 11.5]` would be converted to `[0 .5]` and `[0 .5]`. The RenderMan Shading Language global variable, `time`, is initialized to the value specified via `RiShutter` and so adopting this convention comes at a price - your shaders couldn't tell time! This limitation was lifted with the addition of a new rendering option: `Option "shutter" "float offset" [`*frameoffset*`].`

User Options and Attributes

The RenderMan universe categorizes all aspects of rendering state as either Option, Attribute or Shape. Values that remain constant for the duration of a frame are Options, while Attributes can change from shape to shape within a frame. The aforementioned shutter setting exemplifies a *standard* Option because it has a specific RenderMan request to control it, `RiShutter`, and because it is fundamental to the notion of a camera that the shutter interval remain constant while rendering a frame. `RiSurface` is an example of a standard attribute and allows us to associate a different surface shader with each shape in the frame. Implementation-specific Options and Attributes allow renderers to implement custom extensions to the RenderMan Interface and are expressed via `RiOption` and `RiAttribute`. According to the RI Spec, an implementation can choose to ignore non-standard Options and Attributes and in this way we can have individual RIBs that are targetted to multiple RenderMan renderers.

User Options and Attributes are expressed in terms of `RiOption` and `RiAttribute` but differ from renderer-specific Options and Attributes in that we require compliant renderers to store and correctly scope their values. User Options and Attributes are identified by embedding them in a special namespace, "user", and are made available to RIB archives, shaders and even renderer plug-ins. Pipeline designers were quick to adopt User Options and Attributes as they offered convenient, high-level control over the components of a rendering. For example, it's easy to control shader behavior across a network of RIB archives and shaders by changing a single value defined in a driver RIB file. This simple shader transforms itself according to the value of a user option.

```
surface
mr_jekyl()
{
    uniform string pass;
    option("user:renderpass", pass);
    if( pass == "jekyl") Ci = texture("jekyl.tex");
    else Ci = texture("hyde.tex");
}
```

Conditional RIB and State Variable Substitution

The combination of Conditional RIB and State Variable Substitution enable the possibility of “smart” RIB archives and open up tremendous possibilities to pipeline designers. Conditional RI is expressed with the new requests: `RiIfBegin`, `RiElseIf`, `RiElse`, and `RiIfEnd` and allow the RIB writer to embed different “behaviors” into their RIB. The version of the RIB delivered to the renderer depends on the evaluation of the expressions built into these new conditional requests and the expressions, in turn, are governed by the values of renderer state variables. In addition to User Options and Attributes, standard renderer state variables can be referenced in conditional expressions.

For example, a RIB archive might select different surface shaders depending on which “rendering pass” is active. The driver file might define the current pass with a User Option:

```
Option “user” “string renderpass” [“shadow”]
ReadArchive “myarchive.rib”
```

Then, the archive can use conditionals to express shader association:

```
AttributeBegin
IfBegin “$user:renderpass == ‘shadow’”
    Surface “null”
ElseIf “$user:renderpass == ‘beauty’”
    Surface “rmarble”
Else
    Surface “errorsurf”
IfEnd
Sphere 1.0 -1.0 1.0 360.0
AttributeEnd
```

Inline Archives

Inline Archives round out the collection of smart archive enhancements and have characteristics similar to C pre-processor macros and C++ templates. Inline Archives are defined with: `RiArchiveBegin` and `RiArchiveEnd` and can be implicitly parameterized by rendering state variables. An inline archive is referenced by its archive handle which, like light and object handles, must be unique across all possible RIB archives. When an inline RIB archive is instanced with `RiArchiveInstance`, state variable substitution is performed and the renderer is presented with the results. Inline archives differ from retained geometry in that the instantiation operation is essentially a copying process and not a referencing one. This semantic difference results in far fewer constraints on the contents of Inline Archives than those imposed on RenderMan’s retained objects. Inline archive instantiation occurs at RIB parse time and thus offers advantages over a traditional pre processing solution. As we’ll see, Inline Archives also open up a number of other interesting possibilities.

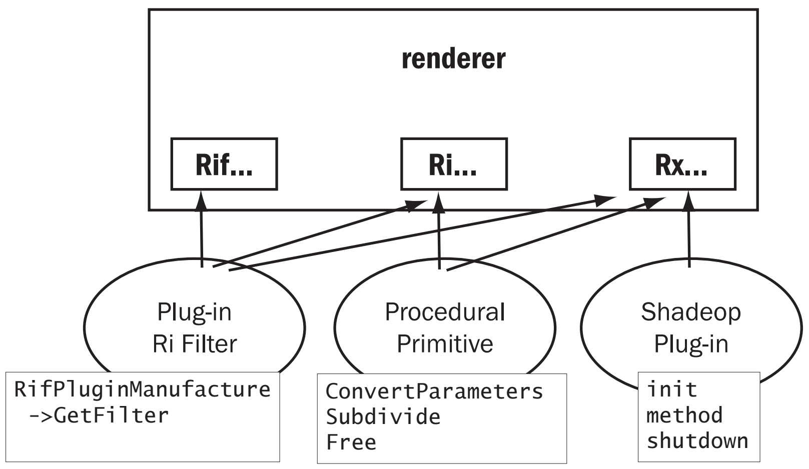
Experiments with Pipeline Plug-ins (2003)

This brings us to the last stop on our “underground tour”. The Smart Archive features are nearly two years old and they’ve generally been well received by RenderMan pipeline designers. Release 12 of Pixar’s PhotoRealistic RenderMan implements a few fixes and improvements to these RIB pipeline features and also introduces two new pipeline features. We’ve found that pipeline designers appreciate programmability much more than your average Joe and so we’ve identified and implemented two new contact-points where pipeline designers and TDs can “do their dirty work”. We introduce two new classes of RenderMan plug-in that offer new opportunities to improve the flexibility, performance and encapsulation of RenderMan based production pipelines.

Plug-in Ri Filters

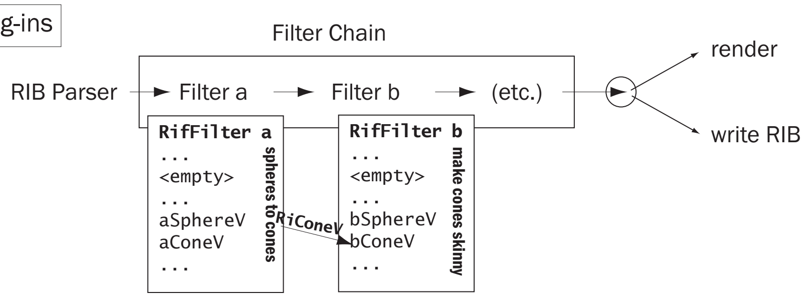
Many of the RIB fixup utilities found in RenderMan production pipelines can accurately be described as RenderMan Interface filters. They parse a RIB stream, search for patterns that match some criteria and replace these patterns with a more preferable, possibly empty, pattern. It’s often convenient to construct *chains* of RIB filters and the result is a powerful back-end RIB processing system that can solve production pipeline problems in a unique way, due largely to its position in the pipeline at the “mouth” of the renderer. To the extent that these filtering systems process data from one on-disk representation to another it can be far from optimal. It’s easy to construct a system that spends more time filtering RIB than rendering. To eliminate the disk I/O overhead and eliminate the burdens associating with parsing RIB, we introduce plug-in Ri filters.

RenderMan Interface for Plug-ins



This figure depicts the contract between the RenderMan renderer and the various plug-in types currently defined. The Ri calls are at the heart of the system and allow procedural primitives and plug-in filters to fundamentally affect the RI stream driving the renderer. The Rx library is provided to allow plug-ins to query rendering state and is available to all plug-in types. For example, RxOption and RxAttribute can be used to lookup values of rendering state variables. The new Rif interface is provided to allow plug-in Ri filters to query and affect the state of the active filter chain. The names in boxes represents the entrypoints required of the various plug-in types by the renderer. Ri filter plug-ins must implement the RifPluginManufacture method and this must construct a C++ object whose class is a derivative of the virtual base-class, RifPlugin. The GetFilter method is required to return a dispatch table of Ri functions and it is through this table that an Ri Filter plug-in can do its work. By implementing a subset of the procedures in the RifFilter structure, the plug-in establishes callbacks that will be invoked as an individual Ri request works its way through the filter chain.

Ri Filtering with Plug-ins



Plug-ins can control whether Ri procedures not implemented by the plug-in result in pass-through or terminate behavior. If a plug-in only implements the RiSphere request, the default filtering mode would determine whether the result was a RIB stream of spheres or everything but spheres.

A trivial filter that converts all spheres to cones would set the default filtering mode to pass-through and would implement a procedure to be called when spheres are encountered. There it would simply invoke RiCone. Because an individual filter has access, via the Rif interface, to the state of the Ri filter chain it can choose to invoke new procedures at the beginning, current, or next point in the filter chain. Here are some snippets from a useful Ri filter plug-in that applies regexp pattern matching to a RIB stream. RIB requests that are bound to an identifier matching the user-provided regular expression are filtered.

First we define our subclass of the RifPlugin virtual base class:

```
class mymatcher : public RifPlugin
{
public:
                      mymatcher(const  char *expr, bool invert);
virtual               ~mymatcher();

virtual RifFilter &   GetFilter(); // returns my filled in dispatch table

private:
    RifFilter         m_rifFilter; // here’s my dispatch table
                                   // we fill it at construction

    void              setName(const char *); // this will be called whenever
                                             // Attribute “identifier” “name” changes.

    std::stack<bool>  m_validityStack; // tracks AttributeBegin/AttributeEnd
                                       // contains bool representing whether the current
                                       // Attribute “identifier” “name” matches the
                                       // expression.

    std::string       m_expression;    // here’s the expression provided by the user
    regex_t           m_regexp;        // a compiled version of the user’s expression
    bool              m_invert;        // should we invert the sense of the test?

    // here are the class methods which will be plugged into the RifFilter
    // dispatch table they will need to access the internals of the current
    // mymatcher object when invoked.

static RtVoid         myAttributeBegin();
static RtVoid         myAttributeEnd();
static RtVoid         myAttributeV(RtToken nm, RtInt n, RtToken tok[], RtPointer val[]);
};
```

Next we implement the entrypoint that the Rif Plugin machinery will invoke to construct an instance of our matcher class. Note that instantiation arguments are provided at the time of construction. For brevity, we omit error checking.

```
RifPlugin *
RifPluginManufacture(int argc, char **argv)
{
    char *pattern =  0L;
    bool invert = false;

    // here we parse incoming arguments looking for -pat regexp and -inv
    for (; argc > 0 && argv[0][0] == ‘-’; argc–, argv++)
    {
       if( !strncmp(“-pat”, argv[0], 4) && argc > 1 )
        {
            argc–;
            argv++;
            pattern = argv[0];
       }
       else if( !strcmp(“-inv”, argv[0]) )
            invert = true;
    }
    if(!pattern) return 0L;
    // all okay here, so construct an instance of mymatcher
    matcher *s = new mymatcher(pattern, invert);
    return s;
}
```

Next, our constructor initializes the members of the filter table with our overrides.

```
mymatcher::mymatcher(const char *expr, bool invert) :
    m_invert(invert),
    m_expression(expr)
{
    // First we initialize our RifFilter structs with the (static) class methods
    m_rifFilter.AttributeBegin = myAttributeBegin;
    m_rifFilter.AttributeEnd = myAttributeEnd;
    m_rifFilter.AttributeV = myAttributeV;

    // here we initialize other state and compile the regular expression
    m_validityStack.push(!m_invert);
    m_rifFilter.Filtering = m_invert ? RifFilter::k_Terminate : RifFilter::k_Continue;
    int err = regcomp(&m_regexp, expr, REG_EXTENDED|REG_NOSUB);
    if(err) ...
}
```

Finally, we define our Ri filter procedures. Note that since the callback functions are static class members we recover the per-object context via a call to `RifGetCurrentPlugin`. The key is to look for changes to the RenderMan attribute “identifier:name”. These come about when the attribute stack is popped or when the attribute is set. These procedures manage the validity stack and apply the pattern matcher, `regexec`, when needed. Whenever the validity state changes, the filter sets its filtering mode accordingly and since we haven’t implemented any other Ri requests, they are subject to this filtering mode.

```
RtVoid
mymatcher::myAttributeBegin()
{
    mymatcher *obj = static_cast<matcher *>( RifGetCurrentPlugin() ); // find mymatcher
object
    obj->m_validityStack.push(obj->m_validityStack.top()); // push the current validity
    RiAttributeBegin(); // pass the RiAttribute call down the chain
}

RtVoid
mymatcher::myAttributeEnd()
{
    mymatcher *obj = static_cast<matcher *>( RifGetCurrentPlugin() );
    obj->m_validityStack.pop();
    if(obj->m_validityStack.top())  // the validity may have changed
       obj->m_rifFilter.Filtering = RifFilter::k_Continue;
    else
       obj->m_rifFilter.Filtering = RifFilter::k_Terminate;
    RiAttributeEnd(); // pass the call down the chain
}

RtVoid
mymatcher::myAttributeV(RtToken nm, RtInt n, RtToken tokens[], RtPointer parms[])
{
    if( !strcmp(nm, “identifier”)  && !strcmp(tokens[0], “name”) )
    {
       mymatcher *obj = static_cast<matcher *>( RifGetCurrentPlugin() );
       obj->setName( ((RtString *) parms[0])[0] );
    }
    RiAttributeV(nm, n, tokens, parms); // pass the call down the chain
}

void
mymatcher::setName(const char *nm) // name has changed, check for a pattern match
{
    bool match;
    if( 0 == regexec( &m_regexp, nm, 1, 0, 0) )
       match = true;
    else
       match = false;
```

```
        m_validityStack.top() = m_invert ? !match : match;
        if(m_validityStack.top())
           m_rifFilter.Filtering = RifFilter::k_Continue;
        else
           m_rifFilter.Filtering = RifFilter::k_Terminate;
    }
```

Now we're ready to run our filter. Ri filter chains must be described outside of the RIB format so we've added some new command-line options to prman to support them.

```
% prman -rif mymatcher.so -rifargs -pat ^her -rifend  example.rib
```

This invocation identifies the location of the Ri filter plug-in and establishes the instantiation arguments for this run of the filter. Here we invoke a single Ri filter but we can easily chain them by supplying multiple -rif blocks. The ordering of the -rif blocks on the command line determines the placement of each filter in the filter chain. Depending on the behavior of your collection of filters, dramatically different results can be obtained simply by reordering the filter chain.

Compared to ad-hoc or home-grown solutions for RIB filtering, this new plug-in type offers a number of advantages. First, the RIB parsing problem is eliminated. The native RIB parser built into the renderer does all the heavy lifting. More importantly, because the filters are expressed as plug-ins, there is no need to put the results onto disk. This virtually eliminates the run-time costs associated with most RIB filters.

There are times when it is desireable to store the filtered RIB stream rather than render it. This turns out to be simple.

```
% catrib -o filtered.rib -rif mymatcher.so -rifargs -pat ^her -rifend  example.rib
```

Now the filtered RIB will magically appear in `filtered.rib` and not rendered. For this version of example.rib:

```
FrameBegin 1
Display  "test.tif" "tiff" "rgba"
Format 512 512 1
Projection "perspective" "fov" [45]
Translate 0 0 10
WorldBegin
AttributeBegin
Attribute "identifier" "name" "my sphere"
Sphere .5 -.5 .5 360
        "constant float foo" [.1]
        "varying float bar" [0 1 2 3]
        "varying float[2] bar2" [0 0 1 1 2 2 3 3]
Attribute "identifier" "name" "his cones"
Cone 1 .5 360
AttributeBegin
Attribute "identifier" "name" "her cylinder and cone"
Cylinder .5 -1 1 360
Cone 2 .5 360
AttributeEnd
Cone 3 .5 360
AttributeEnd
Cone 4 .5 360
Polygon "P" [0 0 0 1 1 1 2 2 2]
WorldEnd
FrameEnd
```

Here’s the result of the above filtering:

```
##RenderMan RIB
version 3.04
FrameBegin 1
Display "test.tif" "tiff" "rgba"
Format 512 512 1
Projection "perspective" "fov" [45]
Translate 0 0 10
WorldBegin
AttributeBegin
Attribute "identifier" "name" ["my sphere"]
Attribute "identifier" "name" ["his cones"]
AttributeBegin
Attribute "identifier" "name" ["her cylinder and cone"]
Cylinder 0.5 -1 1 360
Cone 2 0.5 360
AttributeEnd
AttributeEnd
Cone 4 .5 360
Polygon "P" [0 0 0 1 1 1 2 2 2]
WorldEnd
FrameEnd
```

Note the disappearance of “my sphere” and “his cones”. `mymatcher` is a useful utility that has many of the properties desirable of Ri filter plug-ins. First it’s a very compact plug-in with a tiny, easily maintained codebase. Second its functionality is easy to understand and thus its presence in a chain of filters should offer no surprising side effects. You can even use multiple instances of mymatcher in a pipeline. Today’s pop quiz: how many geometric primitives result from:

```
% catrib -rif matcher.so -rifargs -pat cone -rifend \
         -rif matcher.so -rifargs -pat ^her -inv -rifend
```

In case your imagination hasn’t already run wild, here are a few potential Ri filter applications to consider:

- convert all polygons to subdivision surfaces
- generate level of detail representations of heavy primitives
- fixup texture or shader references
- flatten the transform hierachy

Multipart RIB and Plug-in Instancers

And as if that’s not enough rope for pipeline designers, we introduce another, experimental, pipeline plug-in type that enables the deposition of arbitrary non-RIB data into the RIB stream. Based loosely on the internet’s MIME standard for embedding arbitrary data into email and web pages, the multipart RIB support relies on RIB’s inline archive syntax to both encapsulate arbitrary data and to direct it to the datatype-specific handler for processing. Just as users can express their preference for a particular mpeg viewer to their web browser, your site can configure a RenderMan renderer’s handlers for the content types appearing in your RIB streams.

To describe an arbitrary chunk of data within a RIB, we simply extend the `ArchiveBegin` request to support the “Content-Type” and “Content-Transfer-Encoding” keywords. Currently we support the MIME encodings: “quoted-printable” and “base64” but this is subject to feedback and interest.

```
ArchiveBegin “unique-id”
             “Content-Type” [“application/rib”]
             “Content-Transfer-Encoding” [“quoted-printable”]
ArchiveEnd “unique-id”
```

To request archive processing, we use `ArchiveInstance` with an extended set of arguments that describe instancing options to the plug-in instancer.

```
ArchiveInstance “unique-id” ... (instantiation args)
```

A renderer must define a site-wide and user-specific .mailcap-like table where the mapping between Content-Type and its associated plug-in Instancer is defined. When the renderer’s parser encounters the `ArchiveInstance` request it loads the plug-in designated for the given content type and invokes its exported PluginArchiveInstance procedure. Here’s the function prototype required of all plug-in instancers:

```
int PluginArchiveInstance(
              // the value of “Content-Type” as provided upon archive definition
              const char *contenttype,
              // the data found within the archive definition
              const void *data,
              // the number of bytes of data:
              unsigned size,
              // standard parameter list
              RtInt nargs, RtToken toks[], RtPointer vals[])
```

The same renderer interfaces available to plug-in filters are available to plug-in instancers. Because Inline Archives can be defined and instanced at any (RIB-request aligned) point in the RIB stream the possibilities of plug-in instancers are vast.
Here’s a RIB stream composed almost entirely of tcl:

```
ArchiveBegin “mytclarchive-1” “Content-Type” [“application/tcl”]
 # this is tcl syntax
 Ri::FrameBegin $::frame
 Ri::Projection perspective {{fov 45} }
 Ri::Translate 0 0 10
 Ri::WorldBegin
 switch $::primtype {
  sphere {
    Ri::TransformBegin
    for {set i 0} {$i < 100} {incr i} {
        Ri::Translate 1 0 0
        Ri::Sphere .5 -.5 .5 360
    }
    Ri::TransformEnd
  }
  cone {
    Ri::TransformBegin
    for {set i 0} {$i < 100} {incr i} {
        Ri::Translate 1 0 0
        Ri::Rotate [expr 3.6 * $i] 1 0 0
        Ri::Cone 2 -.5 360
    }
    Ri::TransformEnd
  }
  default {
    Ri::ArchiveRecord comment “Unimplemented primitive $::primtype”
  }
 }
 Ri::WorldEnd
ArchiveEnd “mytclarchive-1”

ArchiveInstance “mytclarchive-1” “int frame” [36] “string primtype” [“sphere”]
ArchiveInstance “mytclarchive-1” “int frame” [37] “string primtype” [“cone”]
```

This example assumes a theoretical tcl binding for RenderMan as well as a plug-in Ri Instancer with a built-in tcl interpretter that converts the Ri:: procedure calls into C Ri calls available to RenderMan plug-ins. The `PluginArchiveInstance` procedure would establish the values of tcl global variables according to the incoming instantiation arguments and then cause the incoming data block to be evaluated in the context of its built-in tcl interpetter. So you think RIB needs more programming features? Just choose your favorite scripting language (and we all have a different one, right?) and follow this recipe. Any renderer that supports Ri Instancer plug-ins will now be able to handle a hybridRIB file built with your language of choice.

Here are other possible applications for Ri Instancer plug-ins:

- render server initialization / cleanup
- asset transport: textures, shaders
- procedural primitive input data delivery
- virus dissemination (be careful!)

Plug-in instancers represent a dramatic shift for RIB and RenderMan pipelines. The possibilities are both exciting and a little frightening. It remains to be seen whether the floodgates they open can be controlled.

Conclusions and Future Directions

The primary goal of the RenderMan Interface is to efficiently describe photorealistic imagery to a 3-D renderer. It's appropriate and not surprising that the grungy plumbing issues associated with the interface take a back seat to the more fundamental, sexy, graphics issues that justify the existance of the RenderMan Interface. Occasionally we step back and reevaluate the usage patterns associated with the Interface and attempt to standardize the plumbing enhancements. Generally these enhancements take the form of experiments or proposals and then we await feedback from the RenderMan community. The success of the shadeop and procedural primitive extensions has made us recognize the need to fully design and promote a standard collection of Rx entrypoints. This work is ongoing.

We've created a forum at **`https://renderman.pixar.com/forum`** for the purposes of collecting feedback and enhancement requests on RenderMan Interface issues. Please contribute your thoughts there!

Acknowledgments

TDs and pipeline designers continually provide us with ideas, inspiration and exasperation to motivate changes in the RenderMan Interface. Pat Hanrahan designed the RenderMan Interface and Tony Apodaca guided RenderMan through adolescence. Pixar's Brian Rosen designed libridiy which was the predecessor and prototype for plug-in Ri filters. Julian Fong, David Laur, Guido Quaroni and Wayne Wooten proofread and critiqued these notes. Pixar's RenderMan Products Team reviewed, critiqued and implemented recent Ri enhancements.

The aswer to today's pop quiz: 2

Chapter 2

Programmable Ray Tracing

David M. Laur, Pixar Animation Studios
dml@pixar.com

2.1 Introduction

This chapter describes the basic ray tracing facilites which have been added to RenderMan and provides an overview of the programming options available to shader writers who want to control ray traced effects. Several examples are presented to illustrate conventional usage, along with a few unconventional ones.

Ray tracing has been used for many great effects over the years, so there is little need to motivate the basic concept. Certainly the idea of the `trace()` operator has been in the RenderMan specification [6] for many years, but production implementations of a hybrid approach are relatively recent. The fundamental expense of the technique compared to the overall efficiency of the REYES architecture for mainline production has often meant that a ray tracer was only brought in for a few special effects. Some caution is still advisable, even with a fully integrated hybrid system. The examples below will show some of the ways in which recent PRMan releases provide both broad and deep **programmable** ray tracing to shader writers. Furthermore, we'll look at a few examples of using ray tracing for collecting data other than reflection and refraction colors.

RenderMan has always had map-based support for rendering reflection and refraction effects, they are critical to many shots. The map-based approach has the great advantage of rendering the final pass very quickly. The maps themselves can be inexpensive to create, depending on the effect and scene. Maps also provide the important artistic benefit of allowing the reflection maps to be created by different processes if necessary. Also, "held" maps can sometimes be reused for several frames, which further amortizes their cost.

There are several reasons to consider ray tracing as an alternative to maps. Ray-tracing can often deliver much greater accuracy than a fixed resolution map, and the "transport-like" nature of rays can lead to better correspondence between effects and real world phenomena. Maps sometimes require sophisticated set up to get right, or multiple maps may be required in situations where a single ray-tracing shader would suffice. Maps are rendered in a separate pass from the main rendering, which in itself isn't usually slower than the corresponding ray tracing, however each pass usually requires its own RIB generation which can be time consuming. There is also the asset management overhead associated with maps, which can sometimes be substantial: they need to be made available on servers, kept up to date when source material changes, shipped between groups with correct file references, etc. The very desirable big-production attributes of maps should always be weighed against the total end-to-end cost of their creation and use. The hours required for a person to set up and manage correct maps are often a lot more valuable for other purposes than the hours spent by a renderfarm server to grind out a similar ray traced effect, especially for one-off images, or during pre-production.

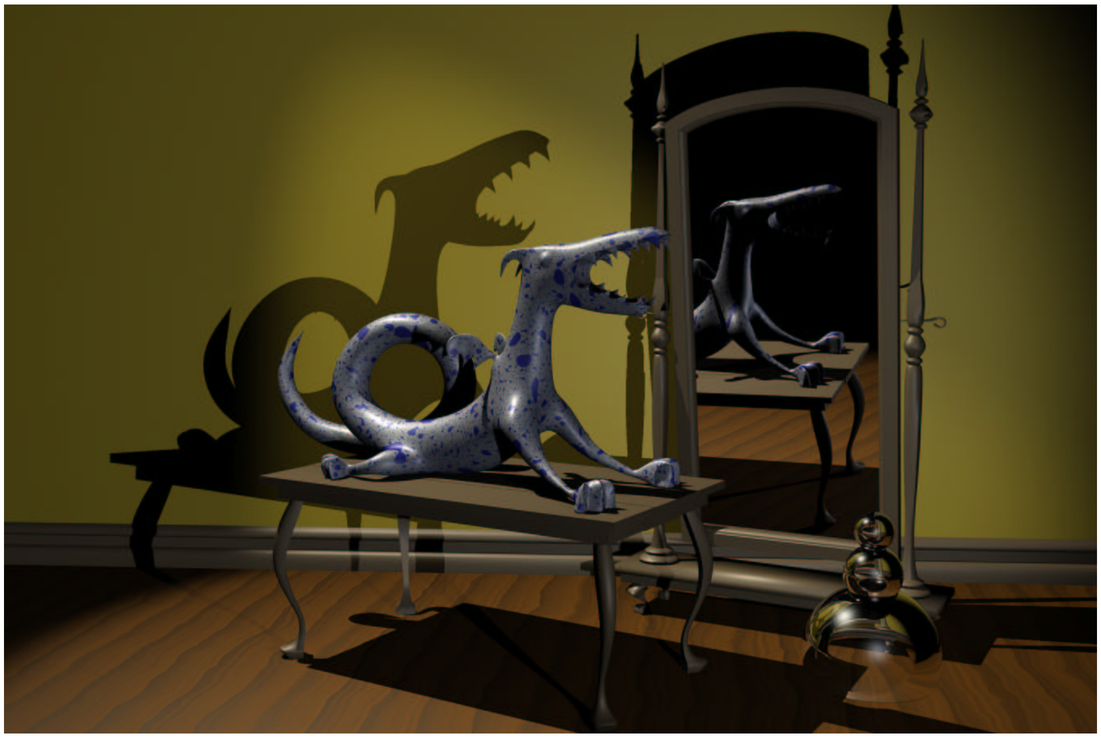


Figure 2.1: An image created in a single pass, rather than using precomputed maps. In this case, ray tracing was used for all of the shadows, reflections, and refraction.

2.1.1 The First Ray is the Really Expensive One

Ray tracing is somewhat at odds with one of original architectural goals RenderMan. In particular, the RenderMan specification was designed with immediate-mode rendering in mind [1, pg60]. The idea being that the scene description should contain no "forward references," the interface hierarchically specifies all of the state which affects a primitive prior to the primitive itself, so each primitive can theoretically be rendered immediately when it is encountered. Furthermore, the memory associated with primitive can be discarded as soon as it is rendered.

Ray tracing, on the other hand, requires a modified approach [1, pg53]. Consider a mirrored sphere that can reflect things which appear after it in the RIB file. It would be very onerous indeed to require all possibly reflected objects to precede the sphere in the RIB stream. In fact it would be impossible to get inter-reflections at all without duplicating geometry. An additional wrinkle arises from the fact that rays can also "see" things that the standard RenderMan pipeline would normally optimize away immediately, [1, pg137] such as objects which end up behind the camera. So ray tracing requires at least some of the geometry to be cached longer than a rendering that isn't doing any tracing. The cost of ray tracing arises from this need for additional memory, plus the inherit computational costs associated with casting a ray through the scene database, resolving intersections along the way.

Shader developers should certainly be aware of these costs when choosing to use ray tracing effects. Careful use of the various ray "visibility" attributes in the RIB file can control how much of the scene actually needs to be cached, and hit tested. These visibility settings also serve the artistic purpose of limiting which objects show up in reflections, etc. The "trace subset" mechanism in PRMan provides even finer-grained control over which primitives can be hit by rays originating from a particular shader. Note that the additional object caching overhead for ray tracing occurs in response to these visibility settings, even if none of the shaders in the scene actually traces a ray! If **any** of the ray visibility modes are enabled, the renderer must assume that tracing is possible and retain the relevant primitives. Even if the first ray doesn't get cast until shading the last pixel of the image, the appropriate scene data needs to be kept around in some form until then.

2.2 Ray Tracing Operators, A Field Guide

2.2.1 `trace`

- **Purpose:** trace a single ray, returning the color of the nearest hit.
- **Typical usage:**
  `color c = trace(P, dir);`
  The direction `dir` is often computed using `reflect()` or `fresnel()`.
- **Why:** easy to understand, trivial to use.
- **Why Not:** single sample only; collects only color from hit shader.
- **Notes:** assign to a `float` to get the hit distance instead:
  `float distance = float trace(P, dir);`

2.2.2 `gather`

- **Purpose:** a looping construct which traces one or more rays, returning arbitrary value from the hit surface, programmable control over multi-sampling.
- **Typical usage:**

```
color c=0, accum=0;
gather ("illuminance", P, dir, cone, samples, "surface:Ci", c) {
    accum += c;
}
Ci = accum / samples;
```

Where `cone` is an angle describing the sampling region, and `samples` is the number of rays used to sample that region.

- **Why:** multi-sampling enables both antialiasing and "blurry" effects; gather can fetch arbitrary values from the hit surface, either geometric properties or output variables from the hit object's shaders; the looping construct allows shaders to accumulate results in a custom manner.

- **Why Not:** modest learning curve

2.2.3 `transmission`

- **Purpose:** Computes the degree of filtering between two points due to intermediate objects. Answers: how much light would be transmitted between the two points?

- **Typical usage:**

```
color t = transmission (Ps, lightpos, cone, samples);
Cl = lightcolor * intensity * t;
```

- **Why:** Provides multi-sample visibility testing, hard or soft shadows; similar to "1$-$*shadow*()" but also useful for other tests of filtering by blockers. More accurate than shadow maps; no separate shadow map pass required; the two test points can be arbitrary whereas shadow maps are computed from a fixed camera position. Computes colored shadows!

- **Why Not:** More expensive than a shadow map lookup.

- **Notes:** Frequently used in lightsource shaders, but can be used in any shader

2.2.4 `environment, shadow`

- **Purpose:** Typically used to read values from pre-computed maps, usually to generate reflections or shadows. When the special keyword "raytrace" is used in place of the texture name, these routines will trace rays instead.

- **Typical usage:** `environment("raytrace", ...); shadow("raytrace", ...);`

- **Why:** Since the texture map name is frequently passed into the shader from the RIB file, existing shaders can be reused as ray tracing shaders, without recompiling, by passing in "raytrace" as the file name.

- **Why Not:** The "native" tracing functions provide more control.

2.2.5 `occlusion`

- **Purpose:** Measures the degree to which a given point is occluded by other objects in the scene, meaning: what percentage of the hemisphere around the current point is blocked? This isn't about shadowing, its about nearness of neighbors. Can be used to fake some types of global illumination without simulating the full light transport in the scene.
- **Typical usage:**
  `Ci = diffusecolor * (1 - occlusion(P, N, samples, ...));`
- **Why:** Has built-in optimizations that make it far more efficient than sampling the hemisphere around a point by other means. "Ambient occlusion" [4] value can add a lot of important subtlety to a frame.
- **Why Not:** Even when optimized, its expensive.

2.2.6 `indirectdiffuse`

- **Purpose:** Computes the indirect contribution to diffuse lighting at a point from nearby objects, such as "color bleeding."
- **Typical usage:** `color indirect = indirectdiffuse(P, N, samples, ...));`
- **Why:** This effect is hard to accomplish in other ways. Can use an irradiance cache to optimize look-ups with interpolation.
- **Why Not:** Can be expensive.
- **Notes:** A lightsource shader which calls indirectdiffuse is a tricky way to contribute indirect color to all existing surface shaders which call diffuse().

2.2.7 `traverse`

- **Purpose:** A looping construct that provides programmable access to the surfaces encountered by a **single ray** as it traverses the scene. Provides a mechanism for "continuing" a ray through transparent objects. Also provides a low-level interface to programmers who want to implement something like their own version of `transmission`.
- **Typical usage:**

```
color c=0, op=0, accumc=0, accumo=0;
traverse ("illuminance", P, dir, "surface:Ci", c, "surface:Oi", op) {
    accumc += c;
    accumo += op;
    if (1 == threshold_exceeded(accumo))
        break;
}
Oi = accumo;
Ci = Oi * accumc;
```

- **Why:** Provides detailed programmability. Also simplifies the process of dealing with partial contribution of transparent objects to reflections.
- **Why Not:** Hard to get all the details right, and the higher level constructs, like transmission and gather, can take efficient shortcuts.

2.3 A Transmission Example: Stained Glass

It is important to remember that `transmission` is concerned with the **filtering** imparted by objects that intervene between its two points of interest. This is fundementally unlike gather and trace which are concerned with the surface characteristics of the *nearest* object.

Frequently the results of a transmission call are are 1.0 or 0.0, that is: light would be completely transmitted (1.0) or completely blocked (0.0). However if the blocking objects are partially transparent, then the transmitted color is fractionally modulated by them.

```
surface stainedglass ( string texturename = "" )
{
    /* a simplistic stainedglass effect,
     * note the decoupling of Oi and Ci
     */
    color x = color texture(texturename);
    Oi = 1 - x;
    Ci = x * x;  /* pleasantly saturated */
}


light transmissionspot (
    float  intensity=1, beamdistribution=2;
    color  lightcolor = 1;
    point  from = point "shader" (0,0,0);
    point  to = point "shader" (0,0,1);
    float  coneangle=radians(30), conedeltaangle=radians(5);
    float samples=1, blur=0, bias = -1;)
{
    uniform vector A = (to - from) / length(to - from);
    uniform float cosoutside= cos(coneangle),
                  cosinside = cos(coneangle-conedeltaangle);
    illuminate( from, A, coneangle ) {
        float cosangle = L.A / length(L);
        color atten = pow(cosangle, beamdistribution) / (L.L);
        atten *= smoothstep( cosoutside, cosinside, cosangle );
        atten *= transmission(Ps, from, "samples", samples,
                        "samplecone", blur, "bias", bias);
        Cl = atten * intensity * lightcolor;
    }
}
```

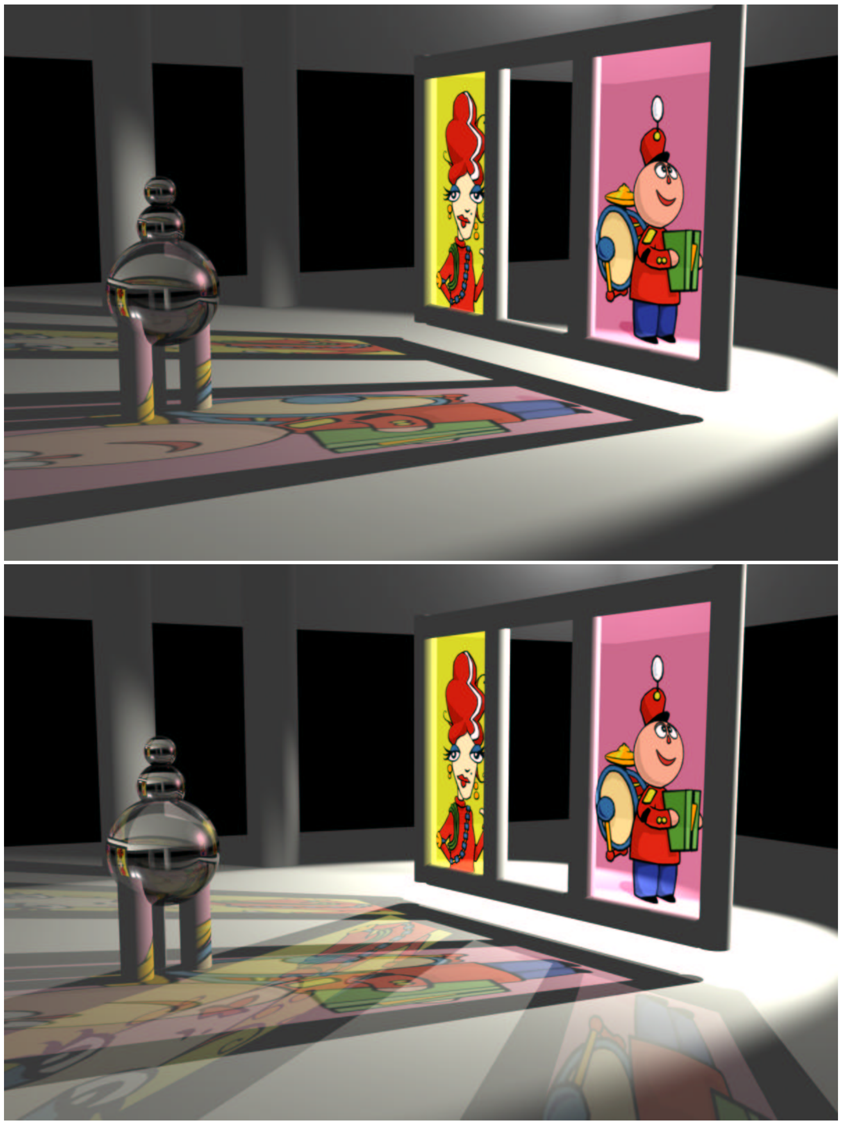


Figure 2.2: Transmission measures the filtering of light between two points. This is often a simple shadow, but color can be modulated too. This image contains a single light source whose shader is making transmission calls to compute the amount of light reaching each surface point. The shader on the two "stained glass" windows simply sets Ci to the texture color and Oi to 1-alpha from the texture map. In the lower image, there are two lights.

2.4 Multi-Sampling

Many of the ray tracing operators describe their sampling region using the half-angle for a cone with apex at P and center along the nominal sampling direction R. Within that sampling cone, a

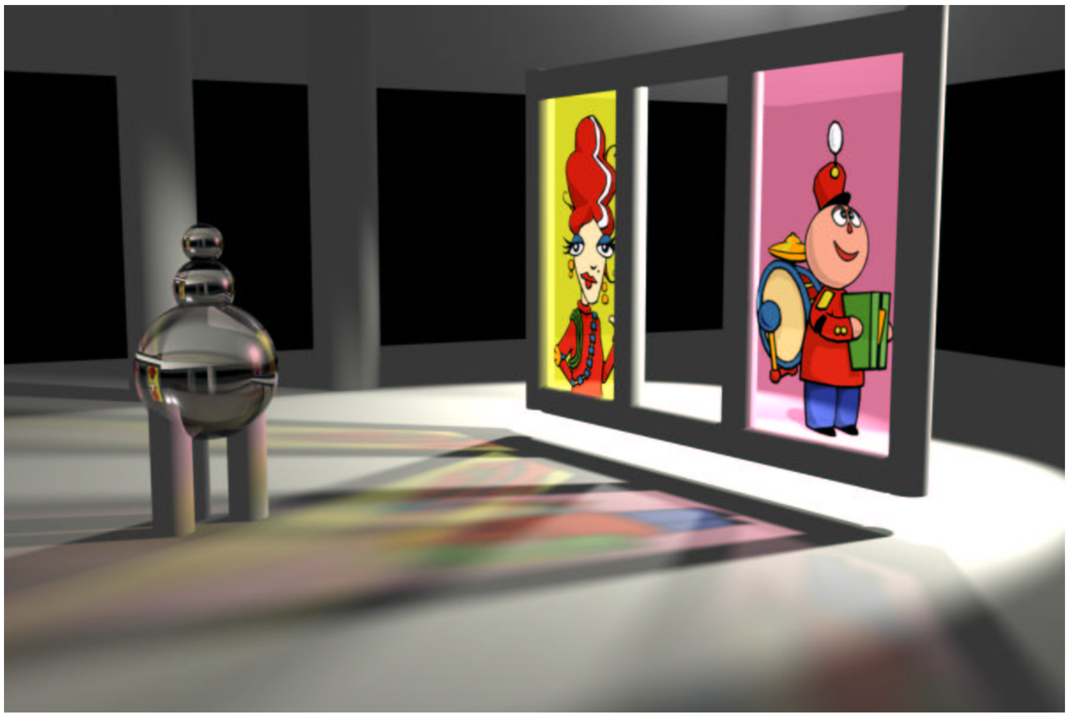


Figure 2.3: Multi-sampled transmission with a small sampling cone. Note the sharpness of the shadows and colors near their ”contact” points, with increasing blur farther away from the cone apex.

number of randomly distributed sample directions are chosen.

```
gather("category", P, R, coneAngle, samples, ...)
```

2.4.1 Multi-Sampling Transmission: Softer Shadows

Recall that `transmission` evaluates the amount of filtering between two points. It can also be used to create soft shadows by sampling a cone instead just a single line between the two points. The idea is to simulate the effect of an area light rather than a point light.

Typical soft shadows are produced by choosing the apex of the sampling cone to be at the surface point (Ps), and the cone should open towards the light source position. Figure 2.3 shows an example of this effect.

Larger cone angles will produce more blur. A fancier simulation might compute the cone angle differently for each value P so that the cone's radius at the light location is the desired apparent size of the light source.

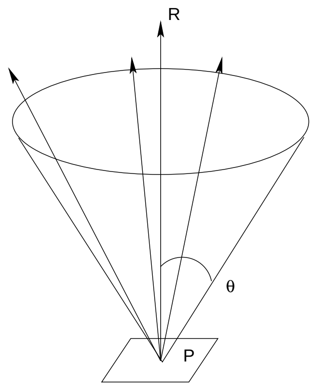


Figure 2.4: A sampling cone described by apex P, cone half-angle, center direction R, and the number of samples.

2.4.2 Gather: Programmable Multi-Sample Surface Queries

While `trace` can be used to shoot a single ray, and `environment` can be used to get a simple average color from many rays, we also need a way to multi-sample which allows us to customize how the results from each ray are treated. Furthermore, we often need to track the case where a sample ray **misses** everything in the scene. The `gather` looping construct addresses these requirements. Here's a little template for using gather:

```
color accumulate = 0;
color hitc = 0;
gather("illuminance", P, dir, cone, samples, "surface:Ci", hitc) {
    /* your code for accumulating hits goes here */
    accumulate += hitc;
} else {
    /* your code for accounting for misses goes here */
}
/* after the loop has run 'samples' times, compute final values */
Ci = accumulate / samples;
```

So, the useful thing about this loop is our ability to apply arbitrary code to each ray's result. We'll see examples of collecting values other than color a little later. Another useful aspect of `gather` is that it automatically picks well-distributed random sampling directions for you from within the specified sample cone. There are currently a couple of built-in sample distribution types to choose from.

2.5 Using `traverse` to Step Through Transparent Objects

One common trick for creating complex scenery for 3D scenes is to simply texture map a painting or photograph of the elements onto a simple polygon "card" or billboard – these might be backdrop matte paintings in some situations, or lots of small ones might be stacked up to create dense shrubbery. Depending on the requirments of the representation, the texture map alpha might be sufficient to represent the extent of the object on the card. In other cases a separate matte texture may be required to represent the object's *coverage*, as distinct from its transparency.

PRMan has a always had the ability to composite objects from the camera's point of view based on their Opacity attribute (`Oi` returned from shaders). However shader writers need make some choices about how these objects should be handled when intersected by secondary rays. One very flexible solution is to make sure that the shaders on the card objects know how to shoot "continuation" rays if they are hit by a ray in an area of the card which is defined to be empty. So rays originating from a mirror would collect the composited result automatically due to the unwinding of the ray tree. This allows the shader on the card to do "anything" it wanted to with its coverage information, but it requires those shaders to be somewhat complex. This solution may also require the default maximum ray depth to be increased so that the continuation rays can make it all the way through the stacked up cards; although the increased depth might have an undesirable effect on other types of rays, so alternatives are needed.

Another solution is to make the card shader very simple:

```
surface texturecard (string texturename = "") {
    Ci = Cs * color texture(texturename);
    Oi = Os * float texture(texturename[3]);
}
```

Then the responsibility for handling ray continuation moves to the originating shader, say a mirror or water surface. One solution is to write a loop around a single-sample call to "gather" which collects opacity and the distance to the hit point (and any other interesting surface values, like coverage, through simple card-shader output variables like those described below). The loop would continue to call gather repeatedly along the same ray direction, while advancing the ray origin on each pass to be just past the previous hit. This technique does produce the correct picture in many cases, but it suffers from several problems, not the least of which is the continual re-traversal of the scene database for each new ray segment. There are also attributes of the original ray such as its "footprint" [2] which get lost when it is chopped up into short segments, and this can have a negative impact on both performance and image quality.

Another approach is to use the `traverse` looping construct, which was designed with this sort of problem in mind. It allows you write your own code for handling hit accumulation along a ray.

```
color c=0,o=0, accumc=0, accumo=0;
traverse("illuminance", P, dir, "surface:Ci", c, "surface:Oi", o, ...)
{
    /* this loop body is called for each surface intersection
     * encountered by this ray, you can early-out via 'break'
     */
    accumo += o;
    accumc += c;

    /* test accumulated opacity (and/or coverage, if available) */
    if (1 == user_opaque_threshold_exceeded(accumo))
        break;
}
```

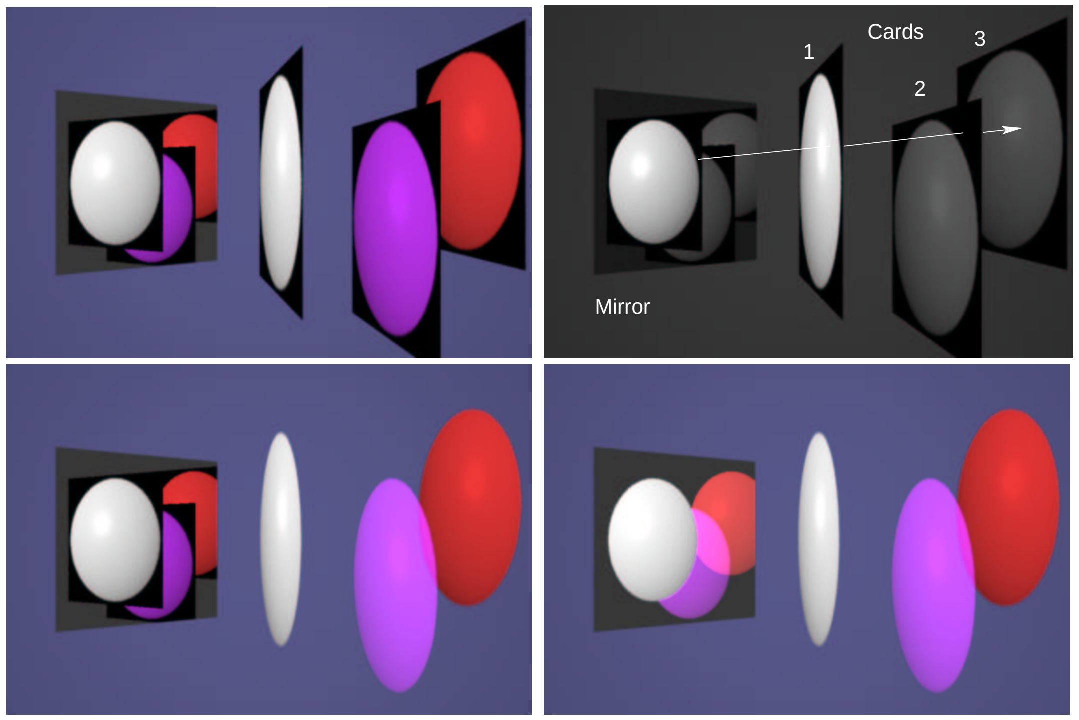


Figure 2.5: Accumulating surface contributions along a ray. The upper left image shows the mirror and three simple polygons which are carrying rgba textures. The upper right illustrates one ray path from the mirror through the cards. The lower left shows the undesireable effect of using a typical `gather` call on the mirror. The lower right shows the desired image, which can be accomplished using several techniques, including `traverse`.

2.6 Adaptive Sampling

One common speed versus quality trade-off provided by some full ray-tracing renderers is to provide some sort adaptive mechanism for reducing the total number of rays cast. The user specifies a minimum acceptable quality or error which then governs sampling density, typically of the primary (camera) rays. The RenderMan primary sampling density is controlled by ShadingRate. When secondary rays are cast from shaders, what sort of opportunities do we have for adaptivity?

It is probably worth prefacing the example below with some cautionary observations. There is a seemingly straightforward approach to adaptivity in which we shoot a few rays, evaluate some function to determine if their value is "good enough" and then either stop or continue to shoot more rays based on the result. Don Mitchell points out [5] that this is fundementally going to generate incorrect results. One really needs to shoot some rays to evaluate the adaptivity criteria, and then shoot a different set of rays to actually collect the desired values. If denser sampling is required then a completely new set of denser rays must be cast. You can't really just shoot a few more rays and claim to have properly refined the previous estimate, because this is basically mixing up two different stratifications of the sampling region. Still, that won't stop us from following the simple-minded strategy in the example below to produce fast "draft" tests, but we'll do so recognizing that we shouldn't rely on this sort of "incremental adaptivity" for final images.

Fully adaptive rendering can be awkward in production because there isn't necessarily an upper bound on the number of samples required, so it can be hard to predict the cost of a rendering. Therefore adaptivity is usually structured as a way to *reduce* the number of samples computed in some regions, with respect to a predefined maximum.

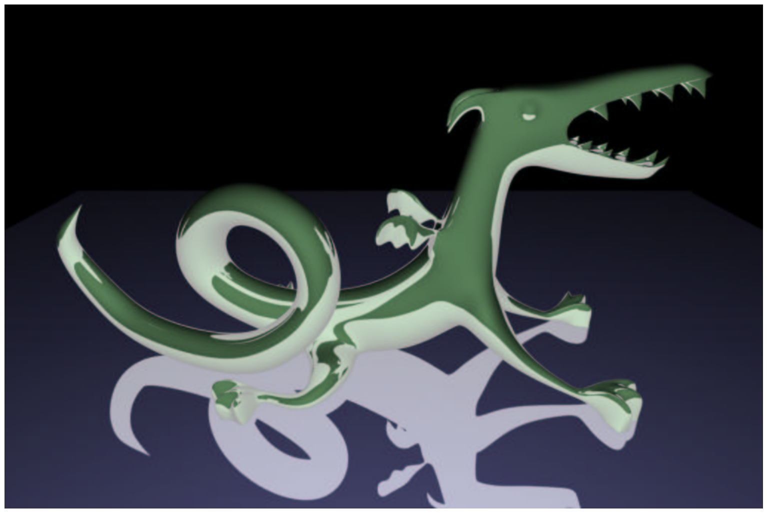


Figure 2.6: Reflection occlusion, shown as the light-colored areas.

Another observation is that for interesting scenes in real productions there often aren't any "constant" areas in which adaptivity can reduce work. You could almost describe SIGGRAPH as the annual meeting of the Society for Adding Subtlety and High Frequency Content to Images. The result is that very often a histogram of adaptive sampling densities used for a particular frame will show spikes at the minimum and maximum allowed values (maximum dominating) and almost nothing in between. So how valuable is a sophisticated adaptive sampling scheme in such a regime really? Some say: "not very" [3]. Instead, ShadingRate, PixelSamples, and the "samples" parameter to `gather` and `transmission` tend to offer more realistic control and predictability in these situations.

However, what if you instead actually have a shot where your ray-shooting surfaces tend to see constant values? Then even a simple-minded adaptivity scheme can help generate frames a lot faster. Remember too that if you are collecting something other than color, for example ray length, there may not be the same high frequency variance in your values as in the color samples. One other important place to find swaths of constant values is in rays which *miss* everything in the scene, they all return the same value!

Consider for example "reflection occlusion" [4] in which the sampling cone can be quite narrow, so it can be well represented by taking relatively few samples from each point. This situation may benefit significantly from a simple adaptive scheme, espcially when measuring self-occlusion of objects which are basically convex.

Figure 2.6 shows reflection occlusion computed on a creature, and the floor. The light-colored

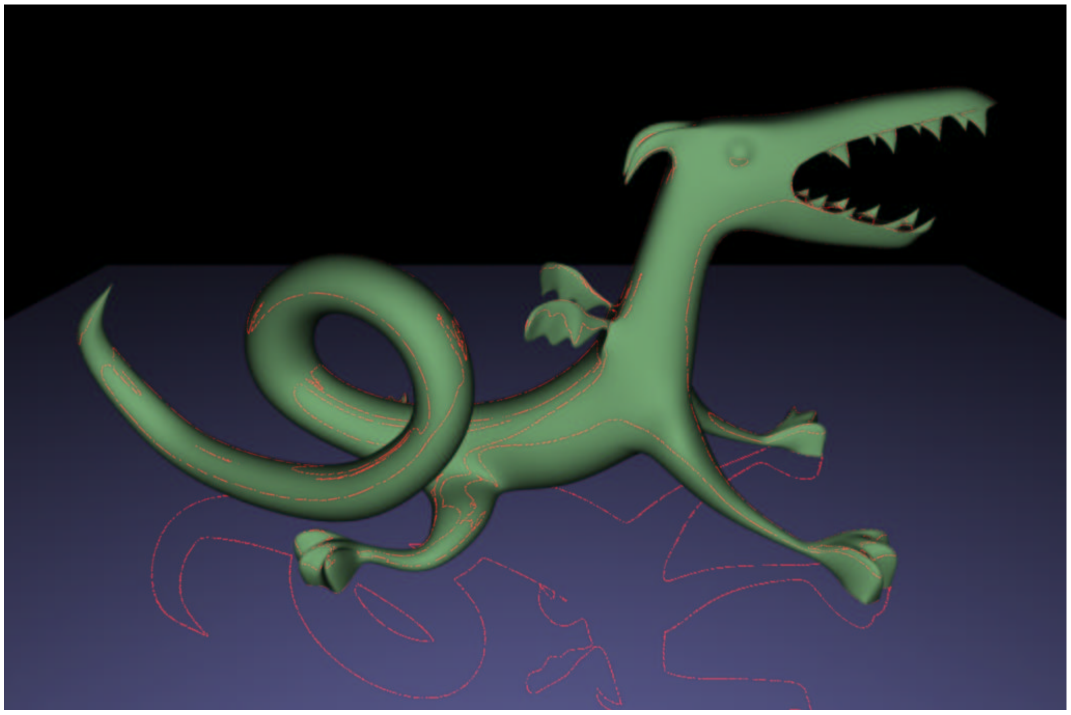


Figure 2.7: Sampling density for a simple adaptive scheme. The maximum sample count was only used when sampling from the points indicated in red.

areas in the image are the areas with high reflection occlusion values. The shader below implements a trivial adaptive-sampling scheme, and it was used to render the same image, using a somewhat ridiculous (minsamples,maxsamples) pair of (4,32). This shader simply introduces a "break" in the gather loop if all of the minsamples agree, otherwise it continues on to maxsamples. It should be straightforward (and advisable!) to make a real shader more sophisticated about its choices.

```
surface reflocc(float minsamples=4, maxsamples=32, visualize=0) {
    vector In = normalize(I);
    vector Nn = normalize(N);
    vector R = reflect(In,Nn);
    float df = In.Nn;
    df = (df*df) / (In.In * Nn.Nn);

    float hits=0, count=0;
    gather("occlusion", P, R, 0, maxsamples) {
        if (count >= minsamples && hits==count)
            break;
        hits += 1;
        count += 1;
    } else {
        if (count >= minsamples && hits==0)
```

```
                break;
            count += 1;
        }
        Oi = Os;
        if (1==visualize) {
            Ci = Oi * mix(df*0.75*Cs, color(1,0.1,0.1), count/maxsamples);
        } else {
            float reflOcclusion = hits / count;
            Ci = Oi * (df*Cs + color(reflOcclusion)) / 2;
        }
    }
```

The adaptive shader produced an image which only varied in a few pixel values compared to the fully maxsampled method. Figure 2.7 is a visualization of the adaptive sampling density.

Due to the simple nature of the objects involved, and the coherent nature of reflection occlusion, it turns out that the shader chooses to use minsamples everywhere except for some very thin regions (shown in red). In these areas the initial set of rays were probing the "silhouette" edges of the figure and so it chose to shoot the full maxsamples. The visualization also suggests that these "contours" themselves might be interesting structures. For this scene, a version of the shader which uses a trivial gather loop that always iterates "maxsamples" times shot 32.3 million rays. The simple adaptive version shot only 5.5 million rays, with a corresponding reduction in rendering time.

2.6.1 Other Kinds of Adaptivity

The built-in `occlusion` function provides both the min-max kind of adaptivity as well as a form of grid adaptivity. It can use a cache of previously computed occlusion values to avoid launching and rays at all at some grid points. It uses various criteria to determine when it can simply interpolate values from the cache.

The new "weight" parameters to the tracing functions provide a mechanism for reducing the number of rays spawned by hits from incoming rays. Rays can be extinguished when their "importance" falls below the given threshold.

Another approach proves that ray tracing and precomputed maps need not be arch-enemies. Consider a map that guides the importance or sampling density of ray tracing. For example a low resolution shadow map might be used to determine if a surface point completely in or out of shadow, or somewhere in between; ray traced shadows could be computed for the grey zone.

2.7 Message Passing Between Shaders

2.7.1 Ray Labels for Categorizing Ray Sources

A shader which is launching rays can specify a "ray label" which gets attached to each ray:

```
gather(..., "label", "some_arbitrary_string");
```

The label can then be queried by shaders on the hit surfaces, using `rayinfo("label", str)`. While these labels might be used to radically alter the behavior of the hit surface shader (consider an "x-ray" mirror), they are often used to modify the expense or quality of the results computed in response to rays from particular sources. The image on the left in Figure 2.8 illustrates a label effect in which the shader on the lower ball produces a greenish color when it is hit by rays from the upper left ball.

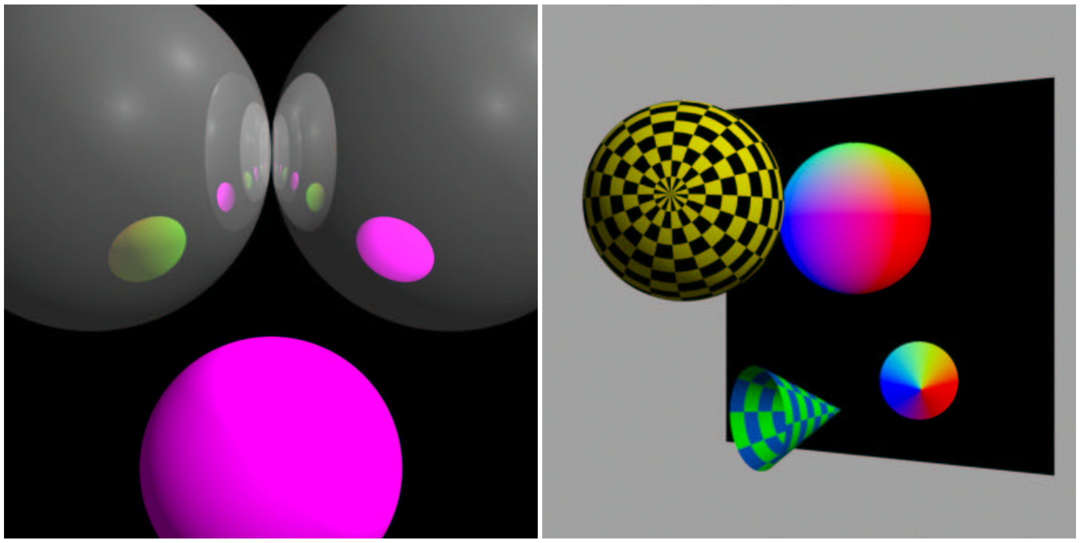


Figure 2.8: Left: the lower ball reports different hit colors based on incoming ray labels. Right: the components of N from the hit surface as RGB.

2.7.2 Fetching Data From Surfaces Hit by Rays

One of the most important aspects of the ray tracing support in PRMan is the ability to collect arbitrary information from the hit surface. Or more precisely, we can collect geometric values from the hit surface, and we can also collect arbitrary values from the surface shader on the hit surface. The right-hand image in Figure 2.8 shows a mirror shader that reflects a visualization of the surface normal. In this case `gather` is used to collect values, including message-passing output variables, from shaders on surfaces hit by rays. The output variable query mechanism can be used to query any of the regular graphics state variables such as `s`, `t`, or `N`.

```
vector hitn=0;
gather(‘‘’’, P, dir, 0, 1, "surface:N", hitn);
```

Note that an alternative nomenclature allows you to specify that these state variables should be collected before the surface shader runs, rather than afterwards as in the case above:

```
vector hitn=0;
gather(‘‘’’, P, dir, 0, 1, "primitive:N", hitn);
```

The “primitive:...” will often run faster than the ”surface:...” when the renderer can determine that the shader at the hit point doesn’t need to be executed at all.

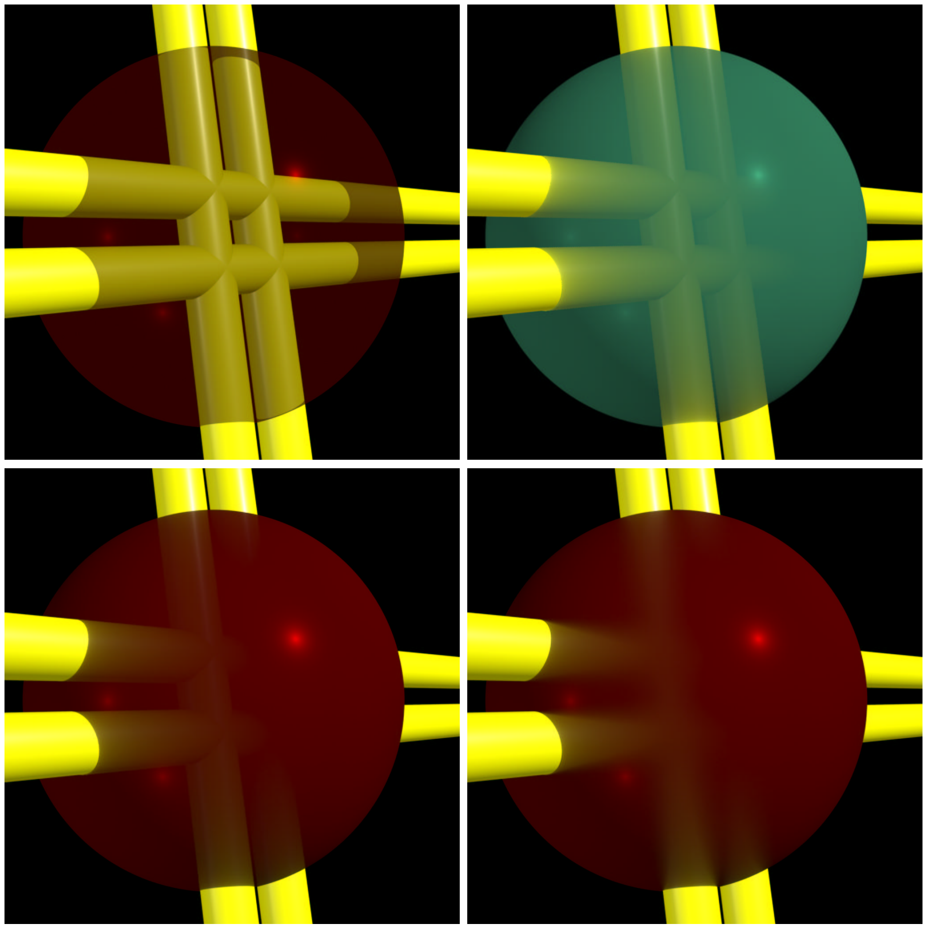


Figure 2.9: Surface color modulated by ray probes. Upper left: standard opacity. Remaining: sphere color modulated by the results of ray tracing into the interior.

2.7.3 Ray Length

The ray length can be a very useful quantity to collect since the distance to the hit surface is often a useful weighting factor. Figure 2.9 shows a sphere which uses ray tracing to modulate its surface color based on what it finds inside. The sphere shader is listed below.

```
surface gumdrop (float Ka=.1, Kd=.2, Ks=.65, roughness=.04,
        maxd=2, falloff=0.955, samples=1, cone=0)
{
    vector Nf = faceforward(normalize(N), I);
    vector In = normalize(I);
    Ci = Cs * (Ka*ambient() + Kd*diffuse(Nf)
                + Ks * specular(Nf, -In, roughness));
    float d = maxd;
```

```
        color c = 0;
        gather("", P, I, 0, samples, "surface:Ci", c, "ray:length", d);
        if (d < maxd) {
         d = pow(d/maxd, log(falloff) / log(0.5)); /* Perlin 'bias' */
            Ci = mix(c, Ci, d);
        }
        Oi = Os;
        Ci *= Os;
    }
```

In this example, multiple samples can be used even if the sampling cone is zero since this will provide antialiasing of the interior results. The upper left image used a standard plastic shader and surface Opacity 0.5, the renderer's regular sample compositing then creates the transparent look. Upper right and lower left illustrate shooting a single ray from each surface point on the sphere to probe the interior, the result modulates the color of the things found by the ray length; the two images simply differ in their modulation function. The lower right image illustrates using a wider sampling cone to produce a murkier look.

Conclusions

RenderMan shaders can use the built-in ray tracing functions to create interesting and complex effects. These facilities are complementary to the traditional map-based approaches for producing reflections and shadows, as well as enabling new kinds effects. In addition to being able to write shaders which rely on the final result of trace operations, the looping constructs, `gather` and `traverse`, provide a way to program custom ray tracing behavior directly in a shader.

Acknowledgements

The integration of flexible, efficient ray tracing into PRMan could only have been accomplished with the hard work of many talented people. Special thanks to Dana Batali, Per Christensen, Julian Fong, and Wayne Wooten for their tireless contributions and design work. Thanks to Tony Apodaca and Mark VandeWettering who lent their experience, insights, and suggestions. Thanks to the RenderMan Products group for all of the support and suggestions, and to the Pixar TDs for keeping us honest, and a little shocked.

References

[1] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan, Creating CGI for Motion Pictures.* Morgan Kaufmann, 2000.

[2] Per H. Christensen, David M. Laur, Julian Fong, Wayne L. Wooten, and Dana Batali. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. *Computer Graphics Forum (Proceedings of Eurographics 2003)*, 22(3), 2003. (To appear).

[3] Larry Gritz. comp.graphics.rendering.renderman, November 1999. 'Re: PixelVariance' ... "Short answer: don't use PixelVariance, it's not a good idea after all.".

[4] Hayden Landis. Production-ready global illumination. In *SIGGRAPH 2002 course notes #16*, pages 87–102. ACM, July 2002.

[5] Don P. Mitchell. Consequences of stratified sampling in graphics. In *Computer Graphics (Proceedings of SIGGRAPH 96)*, pages 277–280. ACM, August 1997.

[6] Pixar Animation Studios. RenderMan Interface Specification version 3.2, July 2000. (https://renderman.pixar.com/products/rispec/).

Chapter 3

Global Illumination and All That

Per H. Christensen, Pixar Animation Studios
per@pixar.com

3.1 Introduction

Global illumination effects include the soft darkening under an object and near edges, color bleeding, etc. These effects are subtle, but important for the realism of an image.

Global illumination effects can be faked with "bounce lights", ie. placing direct light sources at strategic positions in the scene to "fill in" the missing indirect light. Placing bounce lights in a scene requires a lot of painstaking trial and error before a believable result is achieved. (And evaluating many bounce lights at each shading point during rendering isn't cheap either.) We believe that in many cases, simulation of global illumination is the most cost-effective way of obtaining a given effect.

Simulation of global illumination has often been dismissed as being too inflexible and too slow for use in the effects industry. Another objection has been that global illumination algorithms have not been able to handle the insanely complex scenes that are routinely rendered in the industry.

However, in our implementation, the scene causing the indirect illumination can be completely different from the rendered scene, and this gives a high degree of flexibility — there are many knobs to tweak to make the desired adjustments.

While it is true that global illumination computation is slower than a pure scanline rendering, the algorithms continue getting more efficient and the computers continue getting faster.

It is also true that, traditionally, the very incoherent geometry and texture accesses required to compute global illumination have placed severe limits on the scene sizes. However, observations about ray coherency and careful use of multiresolution geometry and texture caches allows us to overcome this limitation; this is described in the course note for SIGGRAPH course 27 [5] and in more detail in a forthcoming paper [4].

Global illumination is not a part of the current RenderMan specification [14]. However, some RenderMan compliant renderers have implemented various flavors of global illumination, for example the Vision system [12, 13] and BMRT [8]. In this note, we describe the implementation of global illumination and related effects in release 11 of Pixar's RenderMan renderer (PRMan). Among the possible effects are super-soft shadows from ambient occlusion, image-based environment illumination (including high-dynamic range images, HDRI), color bleeding, and general global illumination. We also show how to manipulate and tweak global illumination to obtain a desired look, and how to "bake" ambient occlusion and indirect illumination for reuse in an animation.

3.2 Ambient Occlusion

An approximation of the very soft contact shadows that appear on an overcast day can be found by computing how large a fraction of the hemisphere above each point is covered by other objects. This is often referred to as *ambient occlusion*, *geometric occlusion*, or *coverage*. Ambient occlusion has been used for games [19] and for movies such as "Pearl Harbor", "Jurassic Park III", and "Stuart Little 2" [3, 11].

3.2.1 Ambient Occlusion using a Gather Loop

Ambient occlusion can be computed with distribution ray tracing using a gather loop:

```
float hits = 0;
gather ("illuminance", P, N, PI/2, samples, "distribution", "cosine") {
  hits += 1;
}
float occlusion = hits / samples;
```

This gather loop shoots rays in random directions on the hemisphere; the number of rays is specified by the `samples` parameter. The rays are distributed according to a cosine distribution, ie. more rays are shot in the directions near the zenith (specified by the surface normal N) than in directions near the horizon. For each ray hit, the variable `hits` is incremented by one. After the gather loop, occlusion is computed as the number of hits divided by the number of samples. `samples` is a quality-knob: more samples give less noise but makes the rendering take longer.

This gather loop can be put in a surface shader or in a light shader. The following is an example of a surface shader that computes the occlusion at the shading point. The color is bright if there is little occlusion and dark if there is much occlusion.

```
surface occsurf1(float samples = 64)
{
  normal Ns = shadingnormal(N); // normalize N and flip it if backfacing

  // Compute occlusion
  float hits = 0;
  gather ("illuminance", P, Ns, PI/2, samples, "distribution", "cosine") {
    hits += 1;
  }
  float occlusion = hits / samples;

  // Set Ci and Oi
  Ci = (1.0 - occlusion) * Cs;
```

```
    Oi = 1;
  }
```

(A light source shader can do the same; the color is then added in the diffuse loop of surface shaders. See PRMan application note #35 [15] for an example.)

The following rib file contains a sphere, box, and ground plane, all with the `occsurf1` surface shader. There is no light source in this scene.

```
FrameBegin 1
  Format 400 300 1
  PixelSamples 4 4
  ShadingInterpolation "smooth"
  Display "ambient occlusion" "it" "rgba"   # render image to 'it'
  Projection "perspective" "fov" 22
  Translate 0 -0.5 8
  Rotate -40  1 0 0
  Rotate -20  0 1 0

  WorldBegin
    Attribute "visibility" "trace" 1   # make objects visible to rays
    Attribute "trace" "bias" 0.005

    Surface "occsurf1" "samples" 16

    # White ground plane
    AttributeBegin
      Color [1 1 1]
      Polygon "P" [ -5 0 5  5 0 5  5 0 -5  -5 0 -5 ]
    AttributeEnd

    # Red sphere
    AttributeBegin
      Color 1 0 0
      Translate -0.7 0.5 0
      Sphere 0.5  -0.5 0.5  360
    AttributeEnd

    # Multicolored box
    AttributeBegin
      Translate 0.3 0.01 0
      Rotate -30  0 1 0
      Color [0 1 1]
      Polygon "P" [ 0 0 0  0 0 1  0 1 1  0 1 0 ]   # left side
      Polygon "P" [ 1 1 0  1 1 1  1 0 1  1 0 0 ]   # right side
      Color [1 1 0]
      Polygon "P" [ 0 1 0  1 1 0  1 0 0  0 0 0 ]   # front side
      Polygon "P" [ 0 0 1  1 0 1  1 1 1  0 1 1 ]   # back side
      Color [0 1 0]
      Polygon "P" [ 0 1 1  1 1 1  1 1 0  0 1 0 ]   # top
    AttributeEnd
  WorldEnd
FrameEnd
```

The resulting image, shown in figure 3.1(a), is very noisy since `samples` was set to only 16. This means that the occlusion at each point was estimated with only 16 rays. In figure 3.1(b), `samples` was set to 256. As one would expect, the noise is significantly reduced.

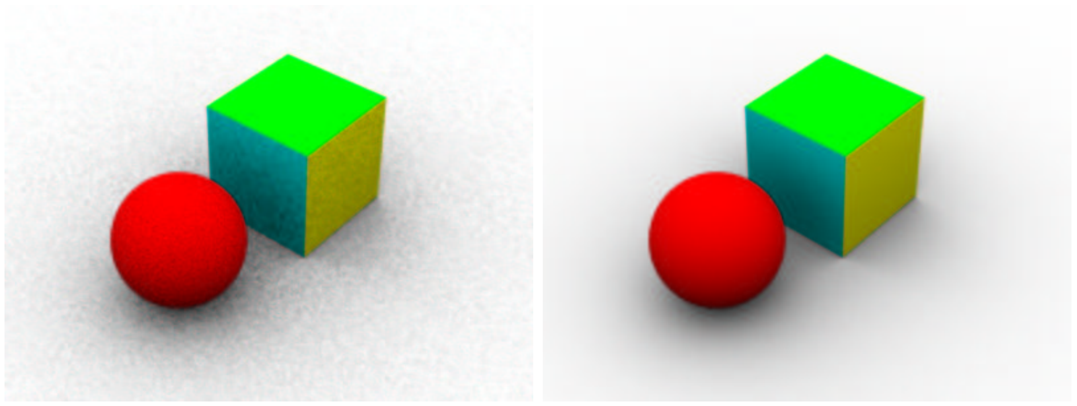


Figure 3.1: Ambient occlusion computed with gather. (a) 16 samples; (b) 256 samples.

3.2.2 Ambient Occlusion using the Occlusion function

Instead of using gather to compute ambient occlusion, it is simpler — and often more efficient — to use the `occlusion` function:

```
occlusion(P, N, samples, ...);
```

`P` is the position where the occlusion should be computed, and `N` is the surface normal at `P`. `samples` specifies how many rays to shoot to estimate the ambient occlusion at point `P`.

The `occlusion` function can be called from surface or light shaders. Here is an example of a surface shader that calls occlusion:

```
surface occsurf2(float samples = 64)
{
  normal Ns = shadingnormal(N); // normalize N and flip it if backfacing

  // Compute occlusion
  float occ = occlusion(P, Ns, samples);

  // Set Ci and Oi
  Ci = (1 - occ) * Cs * Os;
  Oi = Os;
}
```

If the rib file in section 3.2.1 is changed to use surface shader `occsurf2`, and the `Attribute "irradiance" "maxerror"` (which is explained below) is set to 0, images identical to figure 3.1 are computed (in about the same time).

3.2.3 Speedups with the Occlusion Function

The real motivation for using the `occlusion` function is that it can be much faster than using `gather`.

Interpolation

Computing occlusion using `gather` at every shading point is very time consuming. But since occlusion varies slowly at locations far from other objects, it is often sufficient to only sample occlusion at relatively sparse locations, and just interpolate the occlusion at shading points in between. In a typical scene, few parts of the scene are so close to other objects that occlusion has to be computed at every single shading point.

The `occlusion` function exploits this. It only does gathers where it has to; at most locations it can just interpolate occlusion from other gathers nearby. (Technical detail: we interpolate ambient occlusion using occlusion gradients similar to irradiance gradients [18].)

If the `maxerror` attribute is set to 0.5 (the default value), the images in figure 3.2 are computed.

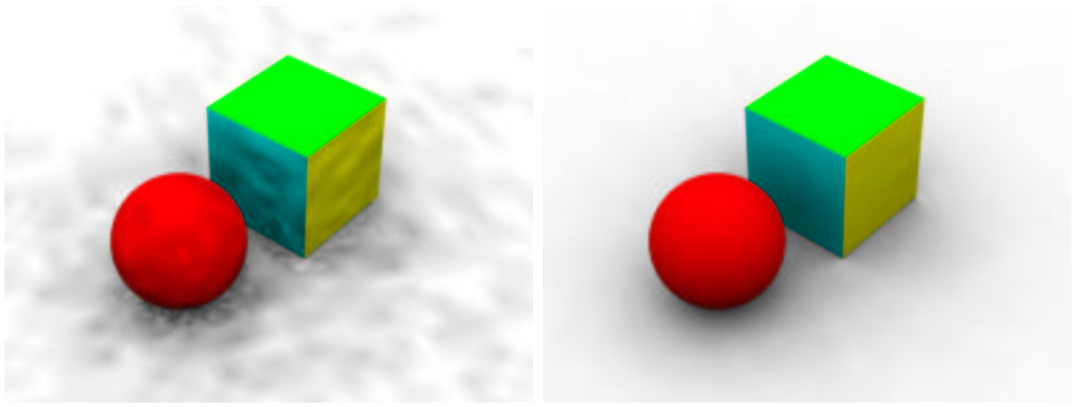


Figure 3.2: Ambient occlusion computed with the occlusion function. (a) 16 samples; (b) 256 samples.

Using `occsurf2` instead of `occsurf1` makes rendering this scene approximately 10 times faster for comparable quality! Notice that low-frequency noise becomes visible when too few samples are used (compare the low-frequency noise in figure 3.2(a) to the high-frequency noise in figure 3.1(a)).

The `occlusion` function uses the `Attribute "irradiance" "maxerror"` as a time vs. quality knob. `maxerror` determines how far a computed occlusion can be used for interpolation of nearby occlusions. (Technical detail: `maxerror` is a multiplier on the harmonic mean distance of the rays that were shot to estimate the occlusion at a point.) If `maxerror` is 0, the occlusion is computed at every shading point, similar to using `gather`. If `maxerror` is very high (like 1000), occlusion is computed sparsely and interpolated over large distances. With intermediate values for `maxerror`, occlusion is computed sparsely far from occluding objects and densely near occluding objects. As mentioned already, the default value for `maxerror` is 0.5.

Adaptive Hemisphere Sampling

Another way to speed up the calculation of ambient occlusion is to adaptively sample the hemisphere. This is controlled by the `occlusion` function's optional parameter `minsamples`. Adaptively sampling of ambient occlusion can give speedups if the scene contains large objects with regular shapes, but gives no improvements if the scene contains lots of tiny objects.

By default, `minsamples` has the same value as `samples`, and the hemisphere is not sampled adaptively. As a rule of thumb, adaptive sampling is not recommended for less than 256 samples (`minsamples` 64), since that gives too few initial samples to base the adaptive sampling on. Using a value of `minsamples` that is less than one quarter of `samples` is not recommended either.

It is twice as fast to compute the image in figure 3.2(b) with adaptive sampling (`samples` 256 and `minsamples` 64) than without adaptive sampling.

3.2.4 Artistic Choices with Ambient Occlusion

There are a few knobs to tweak for artistic choices of the ambient occlusion "look". The `maxdist` parameter determines how far away an object can cause occlusion. Small values make the ambient occlusion a more local effect. The default value is 1e30. The `coneangle` parameter specifies how large a fraction of the hemisphere above a point should be taken into account when computing ambient occlusion. The default value is $\pi/2$, corresponding to the entire hemisphere. `subset` is a name of a subset of objects to consider for ray tracing. The shader `occsurf3` below shows an example of the `maxdist` and `coneangle` parameters.

```
surface occsurf3(float samples = 64, maxdist = 1e30, coneangle = PI/2)
{
  normal Ns = shadingnormal(N); // normalize N and flip it if backfacing

  // Compute occlusion
  float occ = occlusion(P, Ns, samples, "maxdist", maxdist, "coneangle", coneangle);

  // Set Ci and Oi
  Ci = (1 - occ) * Cs * Os;
  Oi = Os;
}
```

Figure 3.3 shows nine variations of `maxdist` and `coneangle`.

3.2.5 More Examples of Ambient Occlusion

Figures 3.4 and 3.5 show some more complicated scenes with ambient occlusion. In image 3.4(a), the areas under the car, as well as the concave parts of the car body, have the very soft darkening that is characteristic of ambient occlusion. (The bumpers and body trim have a chrome material attached to them, so they just reflect light from the mirror direction.) In image 3.4(b), all surfaces (except the glass) are colored by ambient occlusion times the surface color. Figure 3.5 is a money grinder with ambient occlusion.

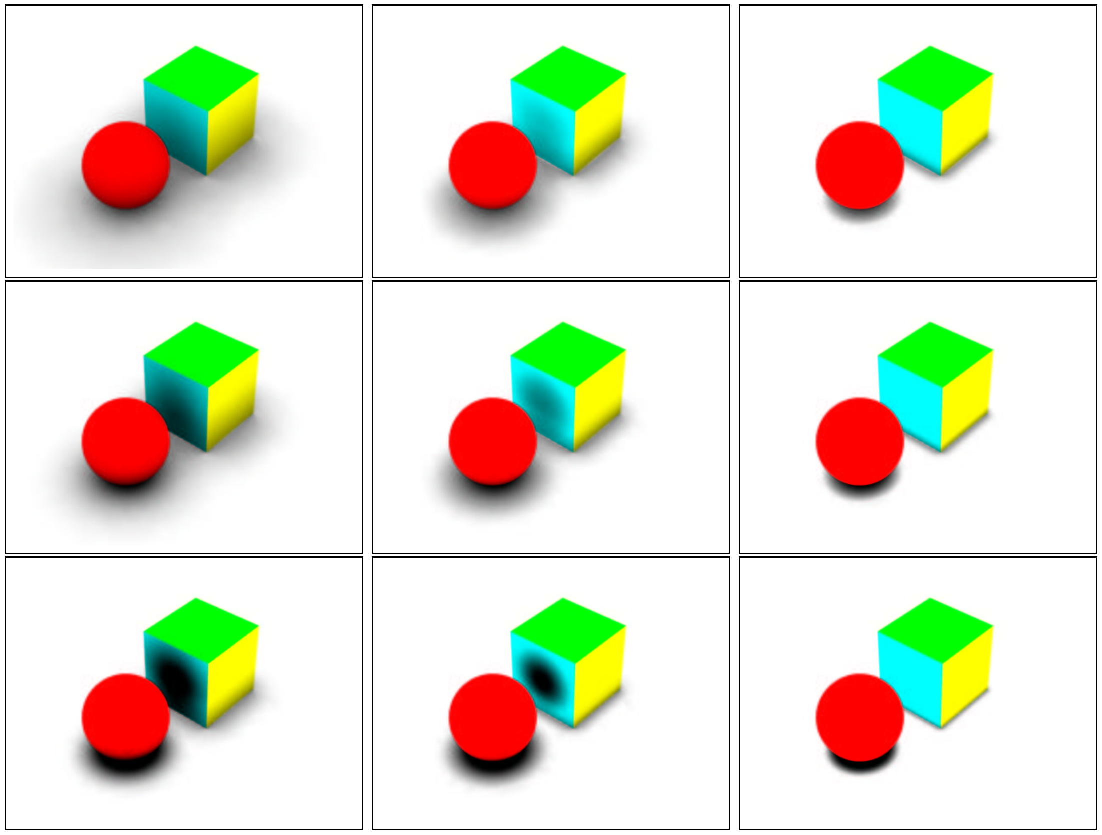


Figure 3.3: Ambient occlusion with different values for coneangle and maxdist. Upper row: coneangle π; middle row: coneangle $\pi/2$; bottom row: coneangle $\pi/4$. Left column: maxdist 1; middle column: maxdist 0.5; right column: maxdist 0.2.

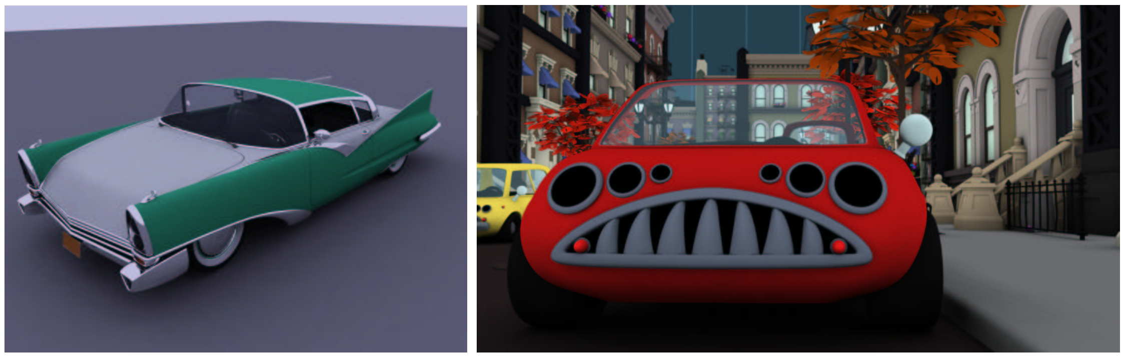


Figure 3.4: More ambient occlusion examples. (a) Al's car. (b) Mike's car.

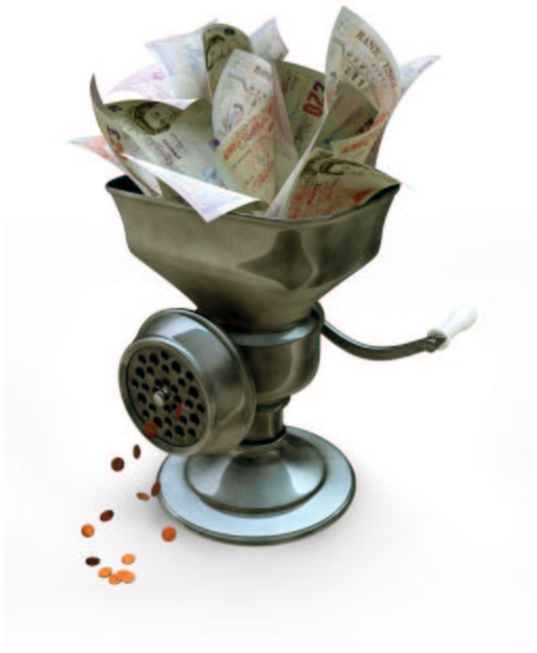


Figure 3.5: Money grinder with ambient occlusion. (Image used with permission of Peter Crowther. Copyright: Peter Crowther Associates.)

3.3 Environment Illumination and Image-Based Illumination

For some looks, we want to illuminate the scene with light from the environment. The environment color can be computed in the shader, but more commonly the environment color is determined from an environment map image; this technique is called *image-based environment illumination* or simply *image-based illumination* or *image-based lighting*. One example of this technique was described in last year's RenderMan course note [3].

To compute image-based illumination, we can either extend the `gather` loop or use more information from the `occlusion` function. There are two commonly used ways of doing the environment map look-ups: either compute the average ray direction for ray misses and use that direction to look up in a pre-blurred environment map, or do environment map look-ups for all ray misses. The examples in this section show all these variations.

Common to all these techniques is that the images used (of course) can contain high dynamic range colors [6, 7].

3.3.1 Image-Based Illumination using a Gather Loop

We first extend the `gather` loop to compute the average ray direction for ray misses. This direction is then used to look up in a pre-blurred environment map. (The environment map should be blurred a lot!) Here is an example of a surface shader that computes image-based illumination this way:

```
surface envillumsurf1(string envmap = ""; float samples = 64)
{
  normal Ns = shadingnormal(N);

  // Compute occlusion and average unoccluded dir (environment dir)
  vector dir = 0, envdir = 0;
  float hits = 0;
  gather ("illuminance", P, Ns, PI/2, samples, "distribution", "cosine",
          "ray:direction", dir) {
    hits += 1;
  } else { // ray miss
    envdir += dir;
  }
  float occ = hits / samples;
  envdir = normalize(envdir);

  // Lookup in pre-blurred environment map
  color envcolor = environment(envmap, envdir);

  // Set Ci and Oi
  Ci = (1.0 - occ) * envcolor * Cs;
  Oi = 1;
}
```

For a first example of image-based illumination, we can replace the shader in the previous scene, and change all objects to be white:

```
FrameBegin 1
  Format 400 300 1
  PixelSamples 4 4
  ShadingInterpolation "smooth"
  Display "environment illumination" "it" "rgba"   # render image to 'it'
  Projection "perspective" "fov" 22
  Translate 0 -0.5 8
  Rotate -40  1 0 0
  Rotate -20  0 1 0

  WorldBegin
    Attribute "visibility" "trace" 1   # make objects visible to rays
    Attribute "trace" "bias" 0.005

    Surface "envillumsurf1" "envmap" "greenandblue.tex" "samples" 16

    # Ground plane
    AttributeBegin
      Polygon "P" [ -5 0 5  5 0 5  5 0 -5  -5 0 -5 ]
    AttributeEnd

    # Sphere
    AttributeBegin
      Translate -0.7 0.5 0
      Sphere 0.5  -0.5 0.5  360
    AttributeEnd

    # Box
    AttributeBegin
      Translate 0.3 0.01 0
      Rotate -30  0 1 0
      Polygon "P" [ 0 0 0  0 0 1  0 1 1  0 1 0 ]   # left side
      Polygon "P" [ 1 1 0  1 1 1  1 0 1  1 0 0 ]   # right side
      Polygon "P" [ 0 1 0  1 1 0  1 0 0  0 0 0 ]   # front side
      Polygon "P" [ 0 0 1  1 0 1  1 1 1  0 1 1 ]   # back side
      Polygon "P" [ 0 1 1  1 1 1  1 1 0  0 1 0 ]   # top
    AttributeEnd
  WorldEnd
FrameEnd
```

(Here greenandblue.tex is a texture that is green to the left and blue to the right, and cyan in between.) All objects are white and pick up color from the environment map. The resulting images (with "samples" set to 16 and 256, respectively) are shown in figure 3.6.

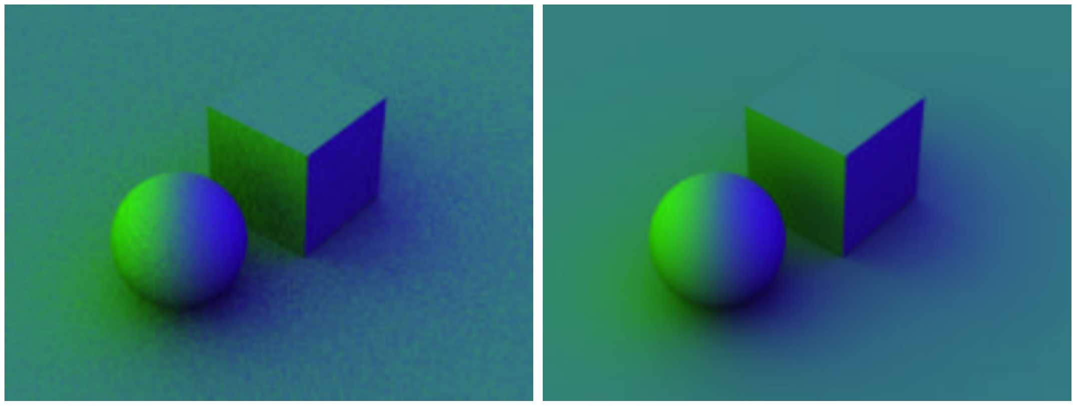


Figure 3.6: Image-based environment illumination using `gather`. (a) 16 samples; (b) 256 samples.

3.3.2 Image-Based Illumination using the Occlusion Function

We can improve the efficiency of image-based illumination by using the `occlusion` function again. The `occlusion` function computes the average unoccluded direction while it is computing the occlusion, and also interpolates those unoccluded directions at the locations where it doesn't actually shoot rays to sample the occlusion. Here is an example of a surface shader that uses `occlusion`'s optional output variable `environmentdir` to compute a rough approximation of the environment illumination.

```
surface envillumsurf2(string envmap = ""; float samples = 64)
{
  normal Ns = shadingnormal(N);

  // Compute occlusion and average unoccluded dir (environment dir)
  vector envdir = 0;
  float occ = occlusion(P, Ns, samples, "environmentmap", envmap,
                        "environmentdir", envdir);

  // Lookup in pre-blurred environment map
  color envcolor = environment(envmap, envdir);

  // Set Ci and Oi
  Ci = (1.0 - occ) * envcolor * Cs;
  Oi = 1;
}
```

Using `envillumsurf2` in the previous scene results in the images shown in figure 3.7.

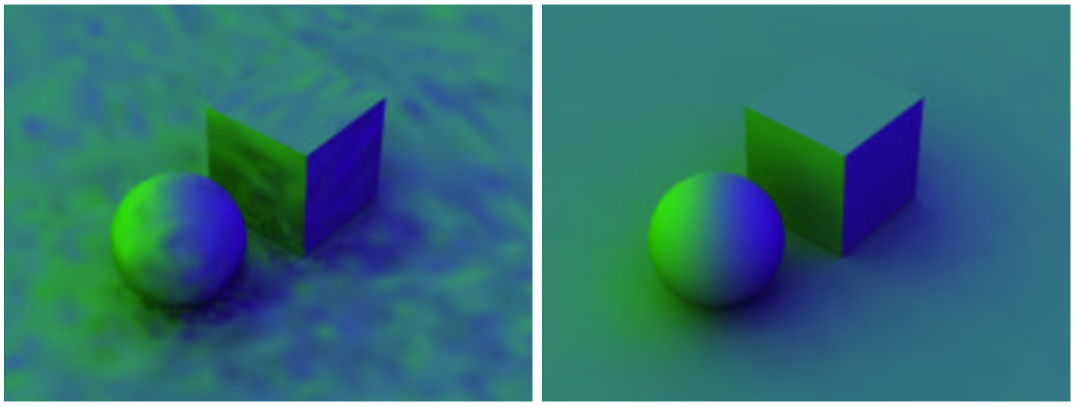


Figure 3.7: Image-based environment illumination using `occlusion`. (a) 16 samples; (b) 256 samples.

Because of the interpolation done in the occlusion function, these images are much faster to compute than figure 3.6. As already mentioned, the environment map image needs to be blurred a lot.

3.3.3 Improved Image-Based Illumination

We can improve the accuracy of the image-based illumination at little extra cost by performing one environment map lookup for each ray miss (instead of averaging the ray miss directions and doing one environment map lookup in the end).

```
surface envillumsurf3(string envmap = ""; float samples = 64)
{
  normal Ns = shadingnormal(N);

  // Compute average color of ray misses
  color sum = 0;
  vector dir = 0;
  gather ("illuminance", P, N, PI/2, samples, "distribution", "cosine",
          "ray:direction", dir) {
    // (do nothing for ray hits)
  } else { // ray miss
    sum += environment(envmap, dir); // lookup in environment map
  }
  color envcol = sum / samples;

  // Set Ci and Oi
  Ci = envcol * Cs;
  Oi = 1;
}
```

In the example in the previous section, this wouldn't make much difference. But if the environment texture had more variation, there could be a big difference between the color in the average unoccluded direction and the average of the colors in the unoccluded directions. An example of a case where this would make a big difference is right under an object: the average direction may be right through the object while the visible parts of the environment form a "rim" around the object.

For improved efficiency, we can again replace the gather loop with a call of the `occlusion` function, and use the optional output parameter `environmentcolor`.

This version of the surface shader looks like this:

```
surface envillumsurf4(string envmap = ""; float samples = 64)
{
  normal Ns = shadingnormal(N);

  // Compute average color of ray misses (ignore occlusion)
  color envcol = 0;
  float occ = occlusion(P, Ns, samples, "environmentmap", envmap,
                        "environmentcolor", envcol);

  // Set Ci and Oi
  Ci = envcol * Cs;
  Oi = 1;
}
```

`environmentcolor` is the average color of rays that didn't hit anything. Each color is found by looking up in the environment map. It pays off to pre-blur the environment map a bit since this gives less noise in the sampling of the environment map.

3.3.4 A High Dynamic Range Image (HDRI) Example

Figure 3.8 shows an example where the environment map used for image-based illumination is an HDR image. The illumination on the diffuse surfaces is computed with the `occlusion` function with the HDR image passed in via the `environmentmap` parameter.

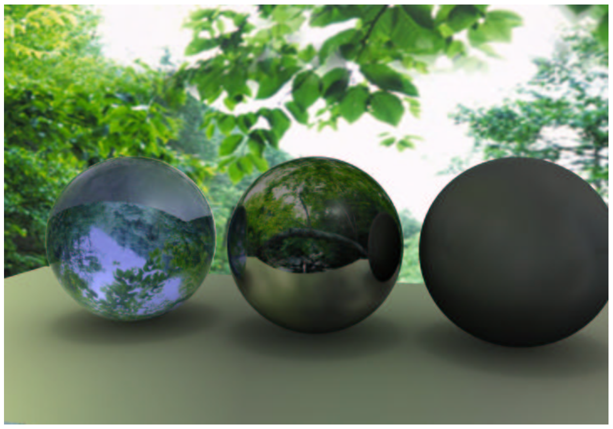


Figure 3.8: Image-based environment illumination with a high dynamic range image.

3.4 Single-Bounce Global Illumination for Color Bleeding

A red carpet next to a white wall gives the wall a pink tint. This is an example of a diffuse-to-diffuse effect called *color bleeding*. In this section we will only look at how to compute a single bounce of diffuse-to-diffuse reflection. In the following section, we consider multiple bounces.

3.4.1 Color Bleeding using a Gather Loop

The illumination at a point on a diffuse surface due to light scattered from other diffuse surfaces and the environment can be found using `gather`:

```
color sum = 0, hitcolor = 0;
vector dir = 0;
gather ("illuminance", P, Ns, PI/2, samples, "distribution", "cosine",
        "ray:direction", dir, "surface:Ci", hitcolor) {
  sum += hitcolor; // add color from hit point
} else { // ray miss
  sum += environment(envmap, dir); // lookup in environment map
}
color indirect = sum / samples;
```

The following shader uses `gather` to compute the soft indirect illumination from other diffuse surfaces and the environment:

```
surface indirectsurf1(float Kd = 1, samples = 16; string envmap = "")
{
  normal Ns = shadingnormal(N);

  // Compute direct illumination
  color direct = diffuse(Ns);

  // Compute soft indirect illumination (if diff. depth < maxdiffusedepth)
  color sum = 0, hitcolor = 0;
  vector dir = 0;
  gather ("illuminance", P, Ns, PI/2, samples, "distribution", "cosine",
          "ray:direction", dir, "surface:Ci", hitcolor) {
    sum += hitcolor; // add color from hit point
  } else { // ray miss
    sum += environment(envmap, dir); // lookup in environment map
  }
  color indirect = sum / samples;

  // Set Ci and Oi
  Ci = Kd * (direct + indirect) * Cs * Os;
  Oi = Os;
}
```

A subtle, but important point: the attribute `maxdiffusedepth` (default value 1) cuts off the rays shot by `gather`. Since the `gather` cone angle is PI/2, the gather rays are classified as diffuse. When one of these rays hits a surface and `indirectsurf1` is evaluated there, that gather will not shoot any rays since the maximum diffuse depth has been reached. This “cropping” of diffuse rays is very important for efficiency; otherwise we would get an explosion in the number of rays.

The following rib file uses `indirectsurf1` to gather diffuse-to-diffuse light. The sphere and box have constant, bright colors, and those colors ”bleed” onto the white diffuse ground plane. The blue color is due to illumination from a blue sky environment map.

```
FrameBegin 1
  Format 400 300 1
  PixelSamples 4 4
  ShadingInterpolation "smooth"
  Display "color bleeding" "it" "rgba"   # render image to 'it'
  Projection "perspective" "fov" 22
  Translate 0 -0.5 8
  Rotate -40  1 0 0
  Rotate -20  0 1 0

  WorldBegin
    Attribute "visibility" "trace" 1   # make objects visible to rays
    Attribute "trace" "bias" 0.005

    # White ground plane with color bleeding onto it
    AttributeBegin
      Surface "indirectsurf1" "envmap" "sky.tex" "samples" 16
      Color [1 1 1]
```

```
        Polygon "P" [ -5 0 5  5 0 5  5 0 -5  -5 0 -5 ]
      AttributeEnd

      # Red sphere
      AttributeBegin
        Surface "constant"
        Color 1 0 0
        Translate -0.7 0.5 0
        Sphere 0.5  -0.5 0.5  360
      AttributeEnd

      # Multicolored box
      AttributeBegin
        Surface "constant"
        Translate 0.3 0.01 0
        Rotate -30  0 1 0
        Color [0 1 1]
        Polygon "P" [ 0 0 0  0 0 1  0 1 1  0 1 0 ]   # left side
        Polygon "P" [ 1 1 0  1 1 1  1 0 1  1 0 0 ]   # right side
        Color [1 1 0]
        Polygon "P" [ 0 1 0  1 1 0  1 0 0  0 0 0 ]   # front side
        Polygon "P" [ 0 0 1  1 0 1  1 1 1  0 1 1 ]   # back side
        Color [0 1 0]
        Polygon "P" [ 0 1 1  1 1 1  1 1 0  0 1 0 ]   # top
      AttributeEnd
    WorldEnd
  FrameEnd
```

The resulting image shown in figure 3.9(a) is very noisy since `samples` was set to only 16. In figure 3.9(b), `samples` was set to 256.

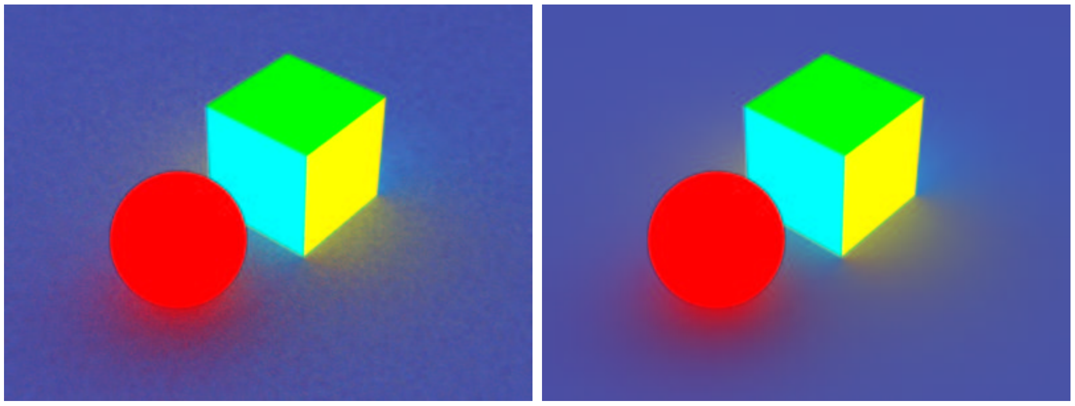


Figure 3.9: Color bleeding computed using `gather`. (a) 16 samples; (b) 256 samples.

3.4.2 Color Bleeding using the Indirectdiffuse Function

Color bleeding can be computed more conveniently (and usually faster) with the `indirectdiffuse` function:

```
indirectdiffuse(P, N, samples, ...);
```

The `indirectdiffuse` function shares many characteristics with the `occlusion` function, and their parameters are nearly identical. The main differences are:

- When a ray shot by `indirectdiffuse` hits an object, the surface shader of that object is evaluated (unless there is a photon map attribute on that object — more on this in section 3.5). Rays shot by `occlusion` do usually not cause the surface shader to be evaluated, and are therefore faster.
- `indirectdiffuse` returns a color while `occlusion` returns a float.

If you want to compute ambient occlusion and indirect diffuse illumination at the same time, the `indirectdiffuse` function has an optional `occlusion` output parameter.

The `indirectdiffuse` function can be called from surface shaders or light shaders. The advantage of doing it in a light shader is that the regular old-fashioned non-globillum-savvy surface shaders such as matte and paintedplastic can still be used. An example of a surface shader calling `indirectdiffuse` is:

```
surface indirectsurf2(float Kd = 1, samples = 16; string envmap = "")
{
  normal Ns = shadingnormal(N);

  // Compute direct illumination
  color direct = diffuse(Ns);

  // Compute soft indirect illumination (if diff. depth < maxdiffusedepth)
  color indirect = indirectdiffuse(P, Ns, samples, "environmentmap", envmap);

  // Set Ci and Oi
  Ci = Kd * (direct + indirect) * Cs * Os;
  Oi = Os;
}
```

If `indirectsurf2` is used instead of `indirectsurf1` in the previous scene, and the `maxerror` attribute is set to 0, similar images are generated (and using similar render time). But using the speedups presented in the following section makes it much faster.

3.4.3 Speedups with the Indirectdiffuse Function

Shooting many rays from every single shading point is very time-consuming. Fortunately, diffuse-to-diffuse effects such as color bleeding usually have slow variation, just as ambient occlusion does. We can therefore determine the diffuse-to-diffuse illumination at sparse locations, and then simply interpolate between these values [18]. This can typically give speedups of a factor of 10 or more.

Illumination can be safely interpolated at shading points that are far away from other geometry, but should not be interpolated at shading points near other geometry. (Fortunately we know how far each illumination sample is from other geometry from the rays that were shot to compute that illumination.) As for occlusion, the tolerance is specified by the `maxerror` attribute (with default value of 0.5). Low values of `maxerror` give higher precision but makes rendering take longer. Setting `maxerror` too high can result in splotchy low-frequency artifacts where illumination is interpolated too far.

We can also use adaptive hemisphere sampling to speed up the computation of indirect illumination. Adaptive hemisphere sampling is controlled by the `minsamples` parameter. Adaptive sampling can give speedups if the scene contains large bright and dark regions, but gives no improvements if the scene contains lots of detail with big changes of color and brightness.

If we use `indirectsurf2` and rerender with the `maxerror` attribute set to 0.5, we get the images shown in figure 3.10.

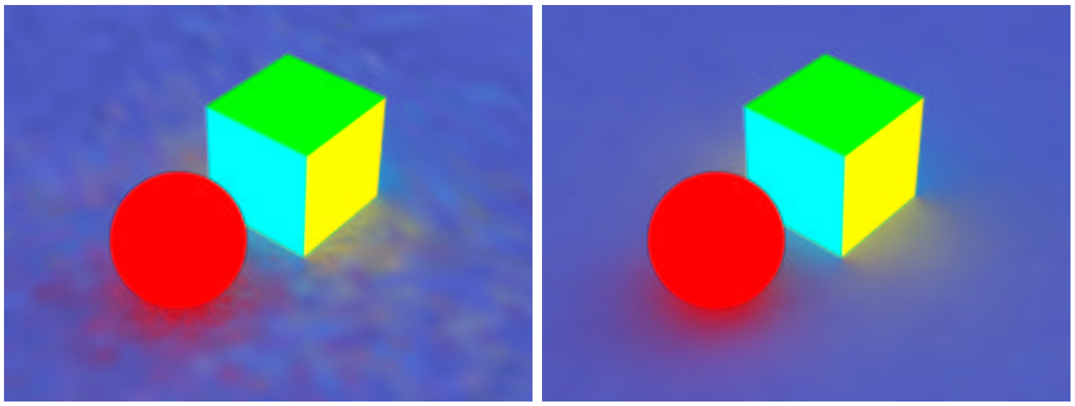


Figure 3.10: Color bleeding computed using the `indirectdiffuse` function. (a) 16 samples; (b) 256 samples.

3.4.4 Another Color Bleeding Example: Matte Box with Spheres

Let's look at another example of color bleeding. A simple rib file describing a matte box with two spheres in it looks like this:

```
FrameBegin 1
  Format 400 400 1
  ShadingInterpolation "smooth"
  PixelSamples 4 4
  Display "cornell box, 1 bounce" "it" "rgba"   # render image to 'it'
  Projection "perspective" "fov" 30
  Translate 0 0 5

  WorldBegin
    LightSource "cosinelight_rts" 1 "from" [0 1.0001 0] "intensity" 4
    # Tiny sphere indicating location of light source ("light bulb")
    # it is invisible to rays
    AttributeBegin
      Surface "constant"
      Translate 0 1 0
      Sphere 0.03  -0.03 0.03  360
    AttributeEnd

    Attribute "visibility" "trace" 1   # make objects visible to refl. rays
    Attribute "trace" "maxdiffusedepth" 1   # one bounce of soft indirect
    Attribute "irradiance" "maxerror" 0.0   # don't interpolate irradiance

    # Diffuse box (with normals pointing inward)
    AttributeBegin
      Surface "indirectsurf2" "Kd" 0.8 "samples" 256
      Color [1 0 0]
      Polygon "P" [ -1 1 -1  -1 1 1  -1 -1 1  -1 -1 -1 ]   # left wall
      Color [0 0 1]
      Polygon "P" [ 1 -1 -1  1 -1 1  1 1 1  1 1 -1 ]   # right wall
      Color [1 1 1]
      Polygon "P" [ -1 1 1  1 1 1  1 -1 1  -1 -1 1 ]   # back wall
      Polygon "P" [ -1 1 -1  1 1 -1  1 1 1  -1 1 1 ]   # ceiling
      Polygon "P" [ -1 -1 1  1 -1 1  1 -1 -1  -1 -1 -1 ]   # floor
    AttributeEnd

    Attribute "visibility" "transmission" "opaque"   # the spheres cast shadows

    # Left sphere (chrome)
    AttributeBegin
      Surface "mirror"
      Translate -0.3 -0.7 0.3
      Sphere 0.3  -0.3 0.3  360
    AttributeEnd

    # Right sphere (diffuse)
    AttributeBegin
      Surface "indirectsurf2" "Kd" 0.8 "samples" 256
      Translate 0.3 -0.7 -0.3
      Sphere 0.3  -0.3 0.3  360
    AttributeEnd
  WorldEnd
FrameEnd
```

Figure 3.11 shows two images of the box. In figure 3.11(a), the box has only direct illumination: the attribute maxdiffusedepth has been changed from 1 (the default value) to 0. This means that no soft indirect diffuse illumination is computed, so the image looks identical to what we would get with the standard matte surface shader. In figure 3.11(b), maxdiffusedepth is 1 and the box has direct illumination and one bounce of indirect illumination. Note the purely indirect illumination of the ceiling and the color bleeding effects.

For historical reasons, this type of illumination computation is sometimes called *final gathering*

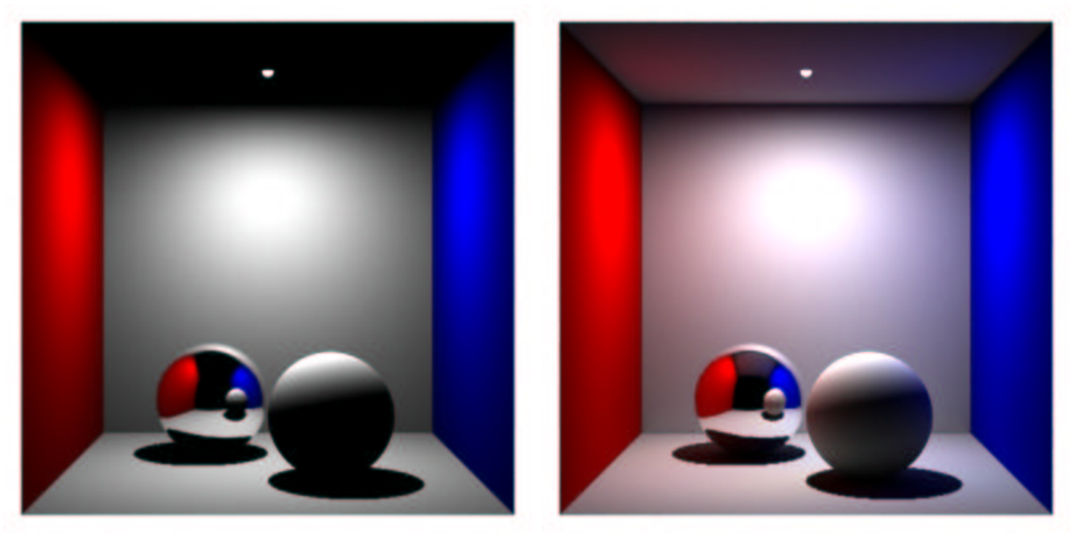


Figure 3.11: Cornell box: (a) direct illumination; (b) direct and single-bounce indirect illumination.

only.

Changing the maxerror attribute to a value larger than 0 enables interpolation of the indirect illumination. This can speed up the rendering significantly. Values for maxerror in the range 0.1 to 0.5 (the default) are typical. Larger values speeds up rendering but can result in splotchy artifacts.

3.4.5 A More Complex Example

Figure 3.12 shows a more complex example. The matte dragons have color bleeding onto them from the sky and ground.

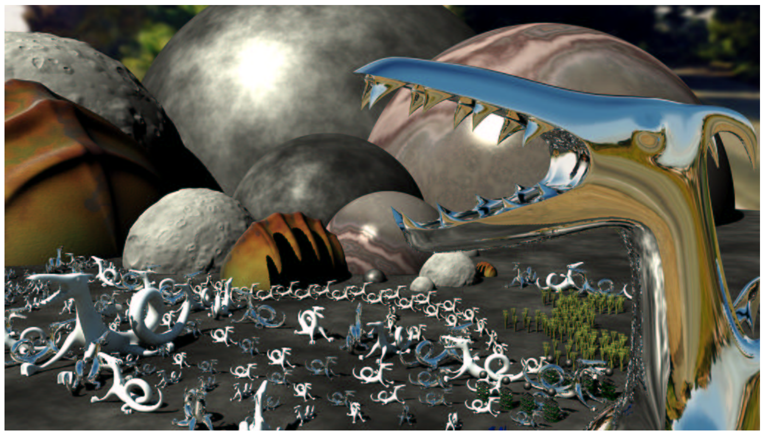


Figure 3.12: Dragon scene with more than 41 million triangles.

3.4.6 Color Bleeding and Efficient Surface Shaders

For performance, it is important that the surface shaders (at the objects the indirect diffuse rays hit) are as efficient as possible. It is often possible to simplify the surface shader computation considerably for diffuse ray hits. For example, it is often possible to reduce the number of texture map lookups, reduce the accuracy of procedurally computed values, etc.

One very important example of this occurs for light shaders that shoot multiple rays to compute antialiased ray traced soft shadows. Such antialiasing is important for directly visible objects or objects that are specularly reflected or refracted, but not necessary when we deal with diffuse reflection. (One might suspect that the same issue would apply to ray traced reflection and refraction rays — shot by `trace` or `"illuminance" gather` loops — but trace and gather already check the ray type and don't shoot any rays if the ray type is diffuse.)

So, when using a shader in a scene with color bleeding computations, it is good shader programming style to check if the ray type is diffuse (or, in general, if the diffuse depth is greater than 0), and use simpler, approximate computations wherever possible, and one shadow ray for ray traced shadows. The ray type can be checked with a call of `rayinfo("type", type)` and the diffuse ray depth can be found with a call of `rayinfo("diffusedepth", ddepth)`.

3.5 Photon Map Global Illumination

In the previous examples, only one bounce of indirect diffuse light was computed. A simple, but very inefficient way to get multiple bounces of indirect illumination is to just increase the maximum diffuse depth and use the same shaders (`indirectsurf1` or `indirectsurf2`).

However, to get multiple bounces in a reasonably efficient way, it is much better to use photon maps. Photon maps are a way of precomputing the indirect illumination; they are described in detail by Jensen et al. [9, 10]. With the photon map method, we get multiple bounces essentially for free since they are precomputed in the photon map; only the last bounce to the eye is expensive — but no worse than in the previous section. In fact, it is often the case that rendering using lookups in a global photon map is faster than computing color bleeding without the photon map. The reason is that the photon map lookups saves us the time spent evaluating the shaders at the ray hit points.

3.5.1 Generating the Global Photon Map

Generating the global photon map is very similar to the generation of a caustic photon map (described in PRMan application note 34 [16]).

Photon maps are generated in a separate photon pass before rendering. To switch PRMan from normal rendering mode to photon map generation mode, the hider has to be set to `"photon"`. For example:

```
    Hider "photon" "emit" 300000
```

The total number of photons emitted from all light sources is specified by the "emit" parameter to the photon hider. PRMan will automatically analyze the light shaders and determine how large a fraction of the photons should be emitted from each light. Bright lights will emit a larger fraction of the photons than dim lights.

The name of the file that the global photon map should be stored in is specified by a `globalmap` attribute. For example:

```
Attribute "photon" "globalmap" "cornell.gpm"
```

If no `globalmap` name is given, no global photon map will be stored. Since `globalmap` is an attribute, different objects in the scene can have different global photon map names. In that case, multiple photon maps are generated. (It is also possible to generate global photon maps and caustic photon maps at the same time: just specify both `globalmap` and `causticmap` attributes. We're not doing that here since this note does not cover rendering of caustics.)

To emit the photons, the light sources are evaluated and photons are emitted according to the light distribution of each light source. This means that for example "cookies", "barn doors", and textures in the light source shaders are taken into account when the photons are emitted. The light sources are specified as usual, for example:

```
LightSource "cosinelight_rts" "light1" "from" [0 0.999 0] "intensity" 4
```

(There are currently some limitations on the distance fall-off for light sources: point lights and spot lights must have quadratic fall-off, solar lights must have no fall-off. This is because photons from a point or spot light naturally spread out so that their density have a quadratic fall-off, and photons from solar lights are parallel so they inherently have no fall-off. We might implement a way around this in a future version of PRMan.)

When a photon has been emitted from a light, it is scattered (ie. reflected and transmitted) through the scene. What happens when a photon hits an object depends on the shading model assigned to that object. All photons that hit a diffuse surface are stored in the global photon map — it doesn't matter whether the photon came from a light source, a specular surface, or a diffuse surface. Photons are only stored on surfaces with a diffuse component; they are not stored on purely specular or completely black surfaces. The photon will be absorbed, reflected or transmitted according to the shading model and the color of the object. The shading model is set by an attribute:

```
Attribute "photon" "shadingmodel" ["matte"|"translucent"|"chrome"|"glass"|
                                   "water"|"transparent"]
```

`Matte`, `translucent`, `chrome`, `glass`, `water`, and `transparent` are the currently built-in shading models used for photon scattering. These shading models use the color of the object (but opacity is ignored). If the color is set to [0 0 0] all photons that hit the object will be absorbed. (In the future, regular surface shaders will also be able to control photon scattering. Regular shaders are of course more flexible, but also less efficient than these built-in shading models.)

Let's look at an example that puts all this together. A simple rib file describing photon map generation in the good ol' matte box with two spheres looks like this:

```
FrameBegin 1
  Hider "photon" "emit" 300000
  Translate 0 0 5

  WorldBegin
    LightSource "cosinelight_rts" 1 "from" [0 0.999 0] "intensity" 4
```

```
      Attribute "photon" "globalmap" "cornell.gpm"
      Attribute "photon" "estimator" 100
      Attribute "trace" "maxspeculardepth" 5
      Attribute "trace" "maxdiffusedepth" 5

      # Matte box
      AttributeBegin
        Attribute "photon" "shadingmodel" "matte"
        Color [0.8 0 0]
        Polygon "P" [ -1 1 -1  -1 1 1  -1 -1 1  -1 -1 -1 ]   # left wall
        Color [0 0 0.8]
        Polygon "P" [ 1 -1 -1  1 -1 1  1 1 1  1 1 -1 ]   # right wall
        Color [0.8 0.8 0.8]
        Polygon "P" [ -1 -1 1  1 -1 1  1 -1 -1  -1 -1 -1 ]   # floor
        Polygon "P" [ -1 1 -1  1 1 -1  1 1 1  -1 1 1 ]   # ceiling
        Polygon "P" [ -1 1 1  1 1 1  1 -1 1  -1 -1 1 ]   # back wall
      AttributeEnd

      # Back sphere (chrome)
      AttributeBegin
        Attribute "photon" "shadingmodel" "chrome"
        Translate -0.3 -0.7 0.3
        Sphere 0.3  -0.3 0.3  360
      AttributeEnd

      # Front sphere (matte)
      AttributeBegin
        Attribute "photon" "shadingmodel" "matte"
        Translate 0.3 -0.7 -0.3
        Sphere 0.3  -0.3 0.3  360
      AttributeEnd
    WorldEnd
  FrameEnd
```

Running this rib file generates a photon map with 279,000 photons; this photon map is shown in figure 3.13(a). Note that the color of each photon is the color it has as it hits the surface, ie. before surface reflection changes its color. Also note that photons are only deposited on diffuse surfaces; there are no photons on the chrome sphere.

As part of the photon map computation, the (diffuse) radiance values are also computed at all photon positions in the global photon map. Computing each of these radiances involves estimating the local irradiance (from the density and power of the nearest photons) and evaluating the shader. A representation of the radiance values in the global photon map is shown in figure 3.13(b). This representation draws a little disc at each photon; the color of the disc is the computed radiance. You might notice that there are little ”cracks” (the little bright spots) between the discs on the matte sphere. These cracks are no cause for concern as this disc representation is only used for visualizing the radiance values, not for rendering. (For rendering we use the nearest photon independent of the disc size.)

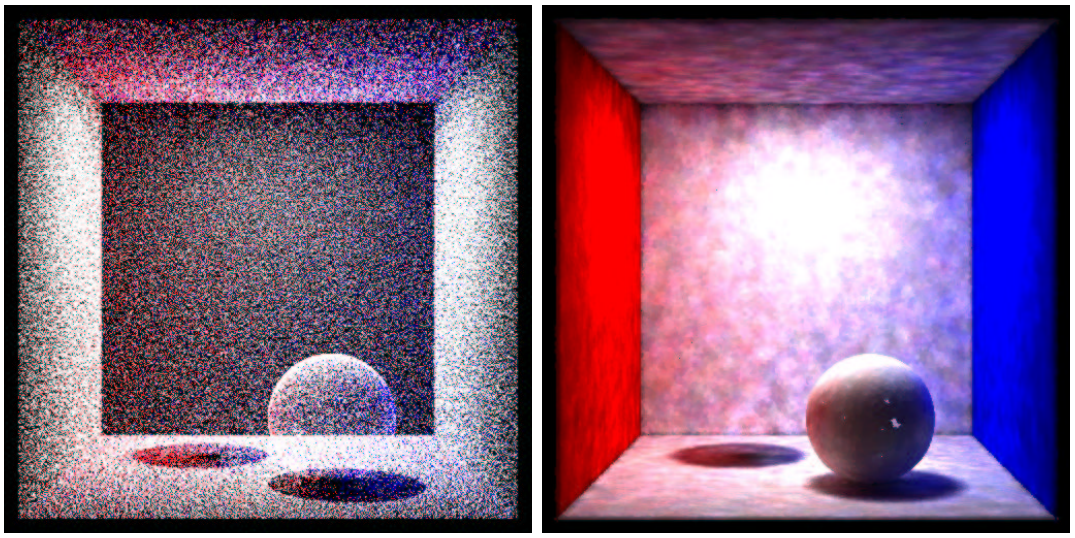


Figure 3.13: Global photon map for Cornell box. (a) photons; (b) irradiance estimates at photon positions.

3.5.2 Displaying Photon Maps

The interactive application *ptviewer* is very useful for scrutinizing the contents of a photon map. It is very often useful to navigate around the photon map (rotate, zoom, etc.) to gain a better understanding of the photon distribution. Ptviewer can show the individual photons in a global photon map as little dots, and can also show the precomputed irradiances at the photon positions as disks.

3.5.3 Rendering Global Illumination

Given a global photon map, there are several ways that shaders can access the indirect color.

First, a shader can get the global illumination color using the `photonmap` function. For example, a surface shader might call like this:

```
Ci = Kd * Cs * photonmap(globalmap, P, N, "estimator", 50);
```

Here `globalmap` is the name of the global photon map, `P` is the position at which to estimate the global illumination, `N` is the surface normal at `P`, and `estimator` specifies how many photons to use for the global illumination estimate. Note that this photon map lookup gives a (rough) estimate of the total illumination at each point, so it would be wrong to add direct illumination (using e.g. an illuminance loop or the diffuse function) to this color. Unfortunately, images rendered in this way have too much noise. Global photon maps are simply too "splotchy" (contain too much low frequency noise) to be acceptable for direct rendering. Two examples are shown in figure 3.14; they use 50 and 500 photons to estimate the illumination, respectively.

As these images show, increasing the number of photons used to estimate the indirect illumination decreases the noise but also blurs the illumination. In order to get crisp, noise-free images,

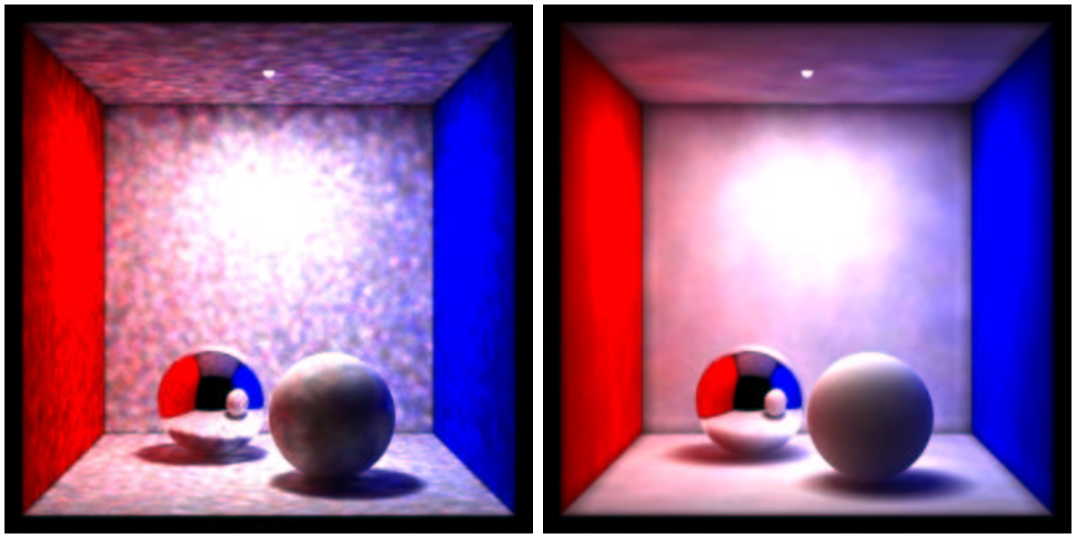


Figure 3.14: Photon map lookups in Cornell box. (a) 50 photons in estimates; (b) 500 photons in estimates.

an extraordinarily large number of photons would have to be stored in the photon map. This is not viable.

Instead, we have to use the `indirectdiffuse` function again — similar to section 3.4. The difference is that when the surfaces have a global photon map attribute:

```
Attribute "photon" "globalmap" "cornell.gpm"
```

the `indirectdiffuse` function picks up photon colors from the global photon map where the rays hit. So for each of the rays shot by the gather loop inside indirectdiffuse, the nearest photon (with appropriate orientation) is found and its precomputed radiance is used. The surface shader is not evaluated at the ray hits, which can be a significant time saving if the surface shader is complicated or if it needs to evaluate many light shaders.

As above, the `indirectdiffuse` function can be in a surface shader or in a light source shader. For example, we can use the same surface shader as in the previous sections:

```
surface indirectsurf2(float Kd = 1, samples = 16; string envmap = "")
{
  normal Ns = shadingnormal(N);

  // Compute direct illumination
  color direct = diffuse(Ns);

  // Compute soft indirect illumination (if diff. depth < maxdiffusedepth)
  color indirect = indirectdiffuse(P, Ns, samples, "environmentmap", envmap);

  // Set Ci and Oi
  Ci = Kd * (direct + indirect) * Cs * Os;
```

```
    Oi = Os;
  }
```

The globalmap attribute is picked up from each surface the light shines on. Here's a rib file for rendering the box using the global photon map cornell.gpm:

```
FrameBegin 1
  Format 400 400 1
  ShadingInterpolation "smooth"
  PixelSamples 4 4
  Display "cornell box, global illum" "it" "rgba"   # render image to 'it'
  Projection "perspective" "fov" 30
  Translate 0 0 5

  WorldBegin
    LightSource "cosinelight_rts" "light1" "from" [0 1.0001 0] "intensity" 4

    # Tiny sphere indicating location of light source ("light bulb")
    # it is invisible to rays
    AttributeBegin
      Surface "constant"
      Translate 0 1 0
      Sphere 0.03  -0.03 0.03  360
    AttributeEnd

    Attribute "visibility" "trace" 1   # make objects visible to refl. rays
    Attribute "photon" "globalmap" "cornell.gpm"
    Attribute "irradiance" "maxerror" 0.0   # don't interpolate irradiance

    # Diffuse box (with normals pointing inward)
    AttributeBegin
      Surface "indirectsurf2" "Kd" 0.8 "samples" 256
      Color [1 0 0]
      Polygon "P" [ -1 1 -1  -1 1 1  -1 -1 1  -1 -1 -1 ]   # left wall
      Color [0 0 1]
      Polygon "P" [ 1 -1 -1  1 -1 1  1 1 1  1 1 -1 ]   # right wall
      Color [1 1 1]
      Polygon "P" [ -1 1 1  1 1 1  1 -1 1  -1 -1 1 ]   # back wall
      Polygon "P" [ -1 1 -1  1 1 -1  1 1 1  -1 1 1 ]   # ceiling
      Polygon "P" [ -1 -1 1  1 -1 1  1 -1 -1  -1 -1 -1 ]   # floor
    AttributeEnd

    Attribute "visibility" "transmission" "opaque"   # the spheres cast shadows

    # Left sphere (chrome)
    AttributeBegin
      Surface "mirror"
      Translate -0.3 -0.7 0.3
      Sphere 0.3  -0.3 0.3  360
    AttributeEnd
```

```
      # Right sphere (diffuse)
      AttributeBegin
        Surface "indirectsurf2" "Kd" 0.8 "samples" 256
        Translate 0.3 -0.7 -0.3
        Sphere 0.3  -0.3 0.3  360
      AttributeEnd
    WorldEnd
  FrameEnd
```

Figure 3.15 shows two images of the box. To the left, the box has only direct illumination and one bounce of indirect illumination (the line `Attribute "photon" "globalmap" "cornell.gpm"` has been commented out — this is the same image in figure 3.11(b)). In the right image, the box has the full global illumination solution.

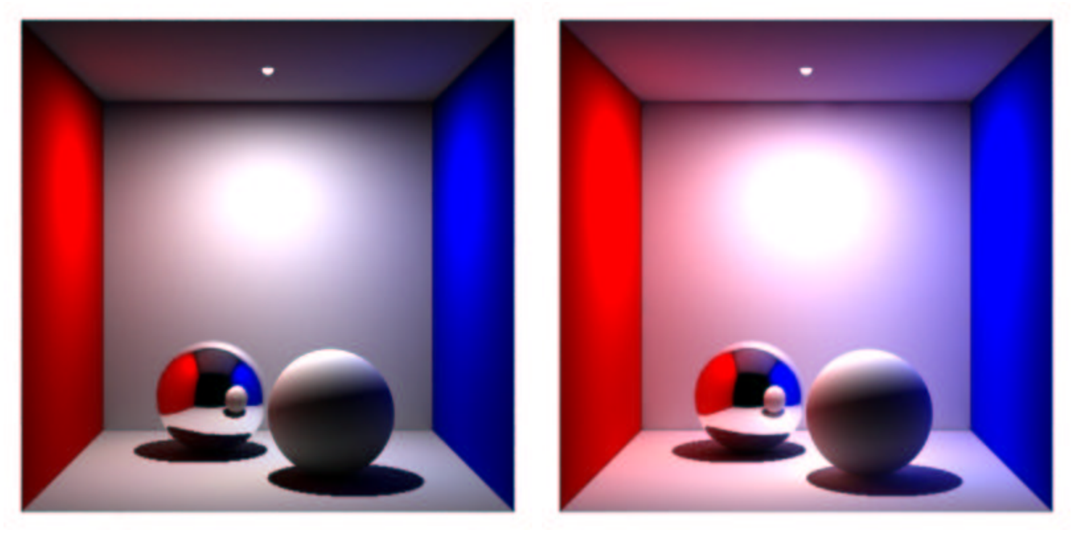


Figure 3.15: Cornell box. (a) single bounce; (b) full global illumination.

Again, changing the maxerror attribute to a value larger than 0 enables interpolation of the indirect illumination. This speeds up the rendering significantly.

3.6 Photosurrealistic Global Illumination

Since photon maps are generated in a separate pass, the scene used for photon map generation can be different from the rendered scene. This gives a lot of possibilities to "cheat" and alter the indirect illumination, similar to the "cheats" traditionally done for direct illumination, shadows, reflection, and refraction [1, 2]. Also, the indirect illumination computed using the photon map can be used just as a basis for the rendering — nothing prevents us from adding more direct lights or darklights ("light suckers") for localized effects, cranking the color bleeding up or down, etc.

Say, for example, that we're in a review meeting with the movie director. We're presenting the beautiful, physically correct global illumination image of our latest shot. (It's probably something more interesting and complex than a Cornell box, but let's stick to that example here for simplicity.) The director says: "Yeah, I like it, it's good, but I want a more blue feel to it. And brighten up that that ceiling. Oh, and while you're at it, I want those shadows on the ground to be more dramatic, but don't change the direct illumination. And I want the reflection of the matte ball in the chrome ball to be larger, but don't move the balls." Translated this means:

- More color bleeding from the blue wall, and less from the red.
- More reflection from the ceiling or an additional light source shining only on the ceiling.
- Move the shadow origin below the real light position.
- Twist the reflection directions from the chrome sphere toward the matte sphere.

The first item requires a change in the scene used for photon map generation. Let's start with that. This only requires changing the colors of the blue and red wall in the rib file for photon map generation:

```
# Matte box
AttributeBegin
  Attribute "photon" "shadingmodel" "matte"
  Color [0.5 0 0]   # dark red
  Polygon "P" [ -1 1 -1  -1 1 1  -1 -1 1  -1 -1 -1 ]   # left wall
  Color [0 0 1]   # bright blue
  Polygon "P" [ 1 -1 -1  1 -1 1  1 1 1  1 1 -1 ]   # right wall
  Color [0.8 0.8 0.8]
  Polygon "P" [ -1 -1 1  1 -1 1  1 -1 -1  -1 -1 -1 ]   # floor
  Polygon "P" [ -1 1 -1  1 1 -1  1 1 1  -1 1 1 ]   # ceiling
  Polygon "P" [ -1 1 1  1 1 1  1 -1 1  -1 -1 1 ]   # back wall
AttributeEnd
```

Running this rib file and inspecting the resulting photon map and irradiance estimates shows that compared to the images in figure 3.13 there are more blue photons and the irradiance estimates have a stronger blue tint.

The second step is to "brighten up the ceiling". We choose to do this by increasing the Kd of the ceiling from 0.8 to 1. This way the color bleeding effects are emphasized. If we had chosen to add an extra light source for direct illumination instead, that would wash out the color bleeding, reducing its visual impact. This is an artistic choice that depends on what the image is supposed to convey.

The third step is to increase the size of the two shadows. This is accomplished by separating the shadow origin from the illumination origin — a standard trick [2] — using the following light source shader:

```
light
cosinelight_tweakedshadow(
  float intensity = 1;
  color lightcolor = 1;
  float falloff = 2;   // default: inverse square fall-off
  point lightfrom = point "shader" (0,0,0);   // light position
  point shadowfrom = point "shader" (0,0,0);   // light position for shadow
  vector dir = (0, -1, 0);
)
{
  illuminate(lightfrom, dir, PI/2) {
    float dist = length(L);
    Cl = intensity * lightcolor * pow(dist, -falloff);   // fall-off
    Cl *= (L.dir) / (length(L) * length(dir));   // cosine term
    Cl *= transmission(Ps, shadowfrom);   // ray traced shadow
  }
}
```

The last step is to make the reflection of the matte sphere appear larger. This is accomplished with the following shader that twists the reflection directions toward a given direction:

```
surface
twistedrefl(float Kr = 1; vector twist = vector(0,0,0))
{
  color Crefl;

  if (N.I < 0) {
    normal Nn = normalize(N);
    vector In = normalize(I);
    vector reflDir = reflect(In,Nn);
    float RdotTwist = reflDir.normalize(twist);
    if (RdotTwist > 0) reflDir += RdotTwist * twist;
    normalize(reflDir);
    Crefl = trace(P, reflDir);
  } else { /* don't reflect inside object (Nn.In > 0 can happen since microp.
              grid shoots rays) */
    Crefl = 0;
  }

  Ci = Kr * Cs * Crefl;
  Oi = 1;
}
```

More ray tracing tricks such as these are described in PRMan application note #36 [17]. Now we are ready to render this “creatively altered” version of the Cornell box. Here is the rib file that puts all these changes together:

```
FrameBegin 1
  Format 400 400 1
  ShadingInterpolation "smooth"
  PixelSamples 4 4
  Display "cornell box, global illum" "it" "rgba"   # render image to 'it'
  Projection "perspective" "fov" 30
  Translate 0 0 5
```

```
    WorldBegin

      # Tiny sphere indicating location of light source ("light bulb")
      # it is invisible to rays
      AttributeBegin
        Surface "constant"
        Translate 0 1 0
        Sphere 0.03  -0.03 0.03  360
      AttributeEnd

      Attribute "visibility" "trace" 1   # make objects visible to refl. rays
      Attribute "photon" "globalmap" "cornell_artistic.gpm"
      Attribute "irradiance" "maxerror" 0.0   # don't interpolate irradiance

      # Diffuse box (with normals pointing inward)
      AttributeBegin
        LightSource "cosinelight_tweakedshadow" "light2"
          "lightfrom" [0 1.0001 0] "shadowfrom" [0 0.4 0] "intensity" 4
        Surface "indirectsurf2" "Kd" 0.8 "samples" 256
        Color [1 0 0]
        Polygon "P" [ -1 1 -1  -1 1 1  -1 -1 1  -1 -1 -1 ]   # left wall
        Color [0 0 1]
        Polygon "P" [ 1 -1 -1  1 -1 1  1 1 1  1 1 -1 ]   # right wall
        Color [1 1 1]
        Polygon "P" [ -1 1 1  1 1 1  1 -1 1  -1 -1 1 ]   # back wall
        Polygon "P" [ -1 -1 1  1 -1 1  1 -1 -1  -1 -1 -1 ]   # floor
        Surface "indirectsurf2" "Kd" 1 "samples" 256
        Polygon "P" [ -1 1 -1  1 1 -1  1 1 1  -1 1 1 ]   # ceiling
      AttributeEnd

      Attribute "visibility" "transmission" "opaque"   # the spheres cast shadows

      # Left sphere (chrome)
      AttributeBegin
        Surface "twistedrefl" "twist" [1 0 -1]
        Translate -0.3 -0.7 0.3
        Sphere 0.3  -0.3 0.3  360
      AttributeEnd

      # Right sphere (diffuse)
      AttributeBegin
        LightSource "cosinelight_rts" "light1" "from" [0 1.0001 0] "intensity" 4
        Surface "indirectsurf2" "Kd" 0.8 "samples" 256
        Translate 0.3 -0.7 -0.3
        Sphere 0.3  -0.3 0.3  360
      AttributeEnd
    WorldEnd
  FrameEnd
```

Figure 3.16 shows the result of these changes along with the original image.

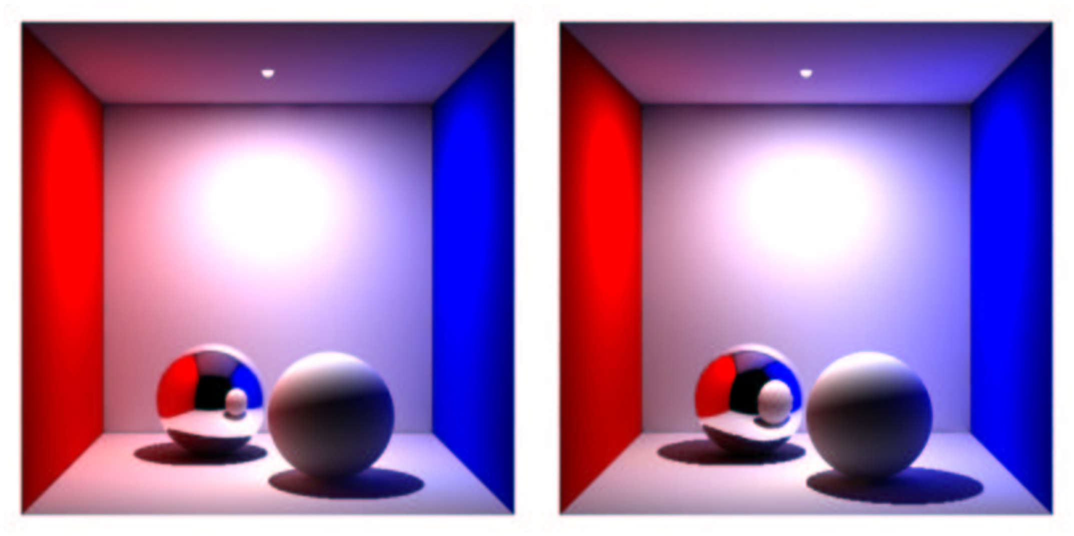


Figure 3.16: Cornell box. (a) original version; (b) surreal version with more blue color bleeding, brighter ceiling, larger shadows and larger reflection.

3.7 Baking of Ambient Occlusion and Indirect Illumination

Sampling the hemisphere above a point is a very expensive operation since it involves the tracing of many rays, possibly many shader evaluations, etc. This is the reason we interpolate ambient occlusion and soft indirect illumination between shading points. To save additional time, we can also reuse computed ambient occlusion and indirect illumination from one frame to the next in animations. This is often referred to as *baking* the ambient occlusion and indirect illumination. So if the ambient occlusion or indirect illumination on some objects does not change significantly, we can compute the ambient occlusion or indirect illumination on these objects once and for all and use it for the entire sequence of frames.

There are three different cases to consider for reuse of cached data: 1) static object, 2) moving object, 3) deforming object. All cases can be combined with a moving camera.

3.7.1 Occlusion and Irradiance Cache Files

Reusing computed irradiance cache data from render to render or from frame to frame is useful to speed up ambient occlusion and indirect illumination (environment illumination, color bleeding, and photon map global illumination). We call such data files *occlusion cache files* or *irradiance cache files*, depending on which data they contain. Figure 3.17 shows an example of an irradiance cache file with 86,000 data points.

There are no restrictions on the naming of these files. One convenient convention is to use suffix `.ocf` for files containing only occlusion information and suffix `.icf` for files containing irradiance (and occlusion) information.

Irradiance cache files can be generated and used explicitly with the `irradiancecache` function, and implicitly with the `occlusion` and `indirectdiffuse` functions (by setting appropriate

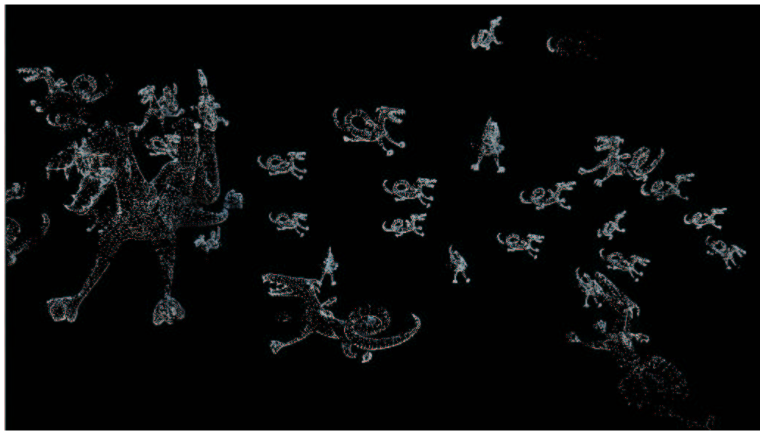


Figure 3.17: Irradiance cache file for dragon scene.

attributes described below).

3.7.2 Generating Occlusion/Irradiance Cache Files

A few "tricks" are necessary to generate an occlusion/irradiance cache for later use in rendering:

1. It is recommended to make all objects semitransparent. This is in order to ensure that occlusion/irradiance is computed on surfaces that are hidden behind other objects for the current viewpoint. (In the future we foresee an attribute for this; then it will no longer be necessary to make the objects semitransparent.)
2. To ensure an even distribution of irradiance data, turn view-dependent gridding off. This is done with the following attribute: `Attribute "dice" "rasterorient" 0`.
3. If the cache file is to be used for other viewpoints, bringing parts that were previously outside the viewing frustum into view, the camera needs to be moved such that those parts of the scene are also visible during cache file generation. Trying to interpolate occlusion/irradiance data from locations where the file does not have any information will give incorrect values (no occlusion and black irradiance).

When generating the irradiance cache with the `occlusion` and `indirectdiffuse` functions (with `filemode "w"` or `"rw"`), the density of the occlusion/irradiance data is determined by the `maxerror` attribute.

It is also possible to generate an occlusion/irradiance cache file directly, this is done with the `irradiancecache` shading function. With the `irradiancecache` function, the data density is determined by the shading rate.

3.7.3 Example of Baking with the Irradiancecache Function

In order to bake ambient occlusion values with the `irradiancecache` function, the following shader can be used. The `irradcachewrite` shader computes ambient occlusion with `gather`, and stores it with irradiancecache("insert", ...).

```
surface
irradcachewrite(string filename = "", coordsys = ""; float samples = 64)
{
  normal Nn = normalize(N);

  // Compute occlusion at P
  float hits = 0;
  gather ("illuminance", P, Nn, PI/2, samples, "distribution", "cosine") {
    hits += 1;
  }
  float occ = hits / samples;

  // Store occlusion in occlusion cache file (in 'coordsys' space)
  irradiancecache("insert", filename, "w", P, Nn, "coordsystem", coordsys,
                  "occlusion", occ);

  // Set Ci and Oi
  Ci = (1 - occ) * Cs * Os;
  Oi = Os;
}
```

The following is a simple rib file for generating three occlusion cache files: wall.ocf, floor.ocf, and hero.ocf.

```
FrameBegin 0

  Format 300 300 1
  ShadingInterpolation "smooth"
  PixelSamples 4 4
  Display "baking_irradcache0" "it" "rgba"
  Projection "perspective" "fov" 30
  Translate 0 0 5

  WorldBegin

    ShadingRate 10   # to avoid too many values in occlusion cache file
    Attribute "dice" "rasterorient" 0   # view-independent dicing
    Attribute "visibility" "trace" 1   # make objects visible to rays

    # Wall
    AttributeBegin
      Surface "irradcachewrite" "filename" "wall.ocf" "samples" 1024
      Polygon "P" [ -1 1 1  1 1 1  1 -1.1 1  -1 -1.1 1 ]
    AttributeEnd

    # Floor
    AttributeBegin
      Surface "irradcachewrite" "filename" "floor.ocf" "samples" 1024
      Polygon "P" [ -0.999 -1 1  0.999 -1 1  0.999 -1 -0.999  -0.999 -1 -0.999]
    AttributeEnd

    # "Hero character": cylinder, cap, and sphere
    AttributeBegin
      Opacity [0.5 0.5 0.5]   # to ensure occlusion is computed behind objects
      Translate -0.3 -1 0   # this is the local coord sys of the "hero"
      CoordinateSystem "hero_coord_sys0"
      Surface "irradcachewrite" "filename" "hero.ocf"
              "coordsys" "hero_coord_sys0" "samples" 1024
      AttributeBegin
        Translate 0 0.3 0
        Scale 0.3 0.3 0.3
        ReadArchive "nurbscylinder.rib"
      AttributeEnd
      AttributeBegin
        Translate 0 0.6 0
        Rotate -90 1 0 0
        Disk 0 0.3 360
      AttributeEnd
      AttributeBegin
        Translate 0 0.9 0
        Sphere 0.3 -0.3 0.3 360
      AttributeEnd
    AttributeEnd

  WorldEnd
FrameEnd
```

Running this rib file produces image 3.18(a) and the three occlusion cache files shown in figure 3.18(b)–(d). (The handy interactive application *ptviewer* was used for visualizing these occlusion cache files.) Note that the occlusion cache images show the occlusion (white meaning full occlusion, black meaning no occlusion) while the image is rendered using 1 minus occlusion.

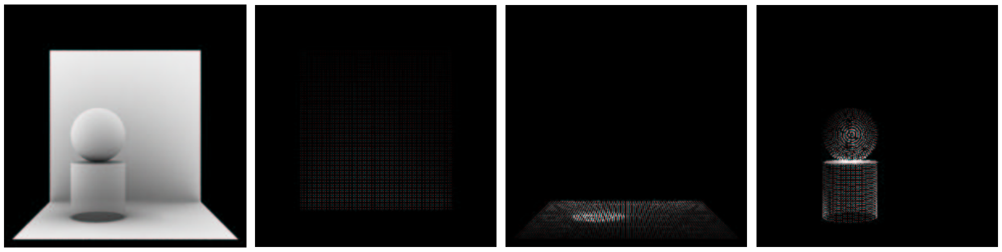


Figure 3.18: Occlusion cache file generation: (a) image; (b) wall.ocf; (c) floor.ocf; (d) hero.ocf.

Next, we want to reuse as much of these occlusion data as possible in an animation where the camera moves and the "hero character" (the cylinder with a sphere on top of it) moves to the right. The back wall is static and the "true" occlusion changes so little that we can reuse the occlusion baked in wall.ocf. The "hero" moves perpendicular to the wall and ground plane, and deforms a bit. His occlusion (in hero.ocf) can also be reused without introducing too much of an error. It is only the floor's occlusion that cannot be reused: moving the hero changes the occlusion on the floor too much — reusing the baked occlusion would leave a black occluded spot on the ground at the place where the hero initially was.

The `irradcacheread` shader reads these data with `irradiancecache("query", ...)`:

```
surface irradcacheread(string filename = "", coordsys = "")
{
  normal Ns = shadingnormal(N);
  float occ = 0;

  // Find nearest precomputed irradiance (use black if none found)
  irradiancecache("query", filename, "r", P, Ns, "coordsystem", coordsys,
                  "occlusion", occ);

  // Set Ci and Oi
  Ci = (1 - occ) * Cs * Os;
  Oi = Os;
}
```

There is a similar shader called `irradcachereadPref` that reads irradiance cache data at reference points. The only difference from the previous shader is that Prefs are passed in as a shader parameter: `varying point Pref = (0,0,0)` and `Pref` is used instead of `P` in the irradiance cache query. The `irradcachereadPref` shader is useful for reusing occlusion values on deforming objects with Pref points.

Here is the rib file for the same scene with the camera moved and the hero character moved and deformed.

```
FrameBegin 3

  Format 300 300 1
  PixelSamples 4 4
  ShadingInterpolation "smooth"
  Display "baking_irradcache" "it" "rgba"
  Projection "perspective" "fov" 30
  Translate 0 0 5
  Rotate -20 1 0 0

  WorldBegin

    Attribute "visibility" "trace" 1
    Sides 1

    # Wall
    AttributeBegin
      Surface "irradcacheread" "filename" "wall.ocf"
      Polygon "P" [ -1 1 1  1 1 1  1 -1 1  -1 -1 1 ]
    AttributeEnd

    # Floor
    AttributeBegin
      Surface "occsurf" "samples" 1024
      Attribute "irradiance" "handle" "floor.ocf"
      Attribute "irradiance" "filemode" ""   # no reuse
      Polygon "P" [ -1 -1 1  1 -1 1  1 -1 -1  -1 -1 -1 ]
    AttributeEnd

    # Cylinder + cap + sphere
    AttributeBegin
      Translate 0.3 -1 0
```

```
        CoordinateSystem "hero_coord_sys3"
        Surface "irradcacheread" "filename" "hero.ocf" "coordsys" "hero_coord_sys3"
        AttributeBegin
          Surface "irradcachereadPref" "filename" "hero.ocf"
                                 "coordsys" "hero_coord_sys3"
          Translate 0 0.3 0
          Scale 0.3 0.3 0.3
          ReadArchive "nurbscylinder_deformed.rib"
        AttributeEnd
        AttributeBegin
          Translate 0 0.6 0
          Rotate -90 1 0 0
          Disk 0 0.3 360
        AttributeEnd
        AttributeBegin
          Translate 0 0.9 0
          Sphere 0.3 -0.3 0.3 360
        AttributeEnd
      AttributeEnd

    WorldEnd
  FrameEnd
```

The rib file `nurbscylinder_deformed` contains a deformed NURBS cylinder with Pref points corresponding to the undeformed NURBS cylinder.

Running the rib file above produces the image in figure 3.19 much faster than if the occlusion values weren't reused.

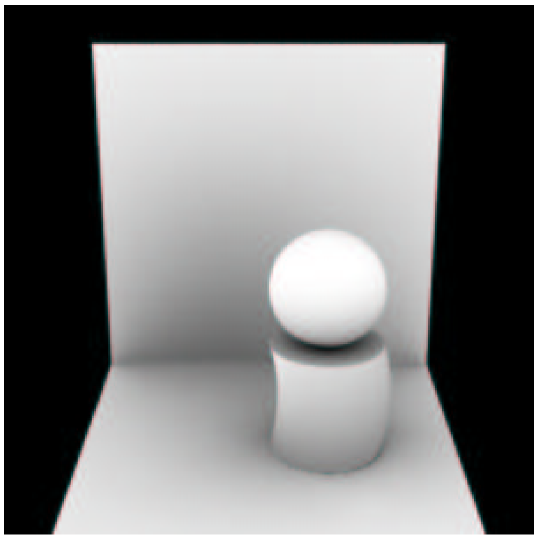


Figure 3.19: Reuse of baked occlusion.

3.7.4 Attributes for Occlusion and Indirectdiffuse

The syntax for specifying a filename that the `occlusion` and `indirectdiffuse` functions should read from and/or write to is:

```
Attribute "irradiance" "handle" "myfilename.icf"
Attribute "irradiance" "filemode" [""|"R"|"r"|"w"|"rw"]
```

Use filemode `"w"` if irradiance computations are to start from scratch and the results should be stored after rendering finishes. Use filemode `"r"` if an existing irradiance cache file should be used to seed the irradiance computations. If an existing irradiance cache file should be used to seed the irradiance computations, and the improved irradiance values should be written out after rendering finishes, use filemode `"rw"`. Filemode `""` means that no file is used to seed the irradiance calculations, and that the irradiance data computed during rendering are not stored in a file at the end.

Filemode `"R"` is a bit special. It is used if we want to rely 100% on the data in an occlusion/irradiance cache file. This makes rendering very fast, but requires that the data in the cache file are sufficiently dense and accurate.

3.7.5 Examples of Baking with the Occlusion and Indirectdiffuse Functions

Consider the previous scene. If the shaders are replaced by shaders calling `occlusion`, and the irradiance handle and filemode attributes are set to appropriate values, we will get similar results.

The occlusion data in the occlusion cache files are distributed differently because the occlusion function computes ambient occlusion sparsely where there is low variation, while gather is computed at every shading point. The rendered images look approximately the same.

A more interesting test is to replace the above shaders with shaders that bake and reuse irradiance. And let's add a large bright green patch to the scene. All other surfaces are write.

The shader `indirectsurf1sided` computes indirect illumination using the `indirectdiffuse` function:

```
surface indirectsurf1sided(float samples = 64, maxdist = 1e30;
                           string coordsystem = "")
{
  normal Nn = normalize(N);

  color indirect = indirectdiffuse(P, Nn, samples, "maxdist", maxdist,
                                   "coordsystem", coordsystem);

  Ci = indirect * Cs * Os;
  Oi = Os;
}
```

The reason we use a special "one-sided" shader for this scene deserves an explanation. Since we want to generate irradiance values also on surfaces that face away, we cannot set `Sides` to 1. We cannot use the `faceforward` function either, since that would compute irradiance on the inside of the most distant half of the sphere and cylinder. So we have to respect the orientation of the surfaces in the scene. (Other scenes have different characteristics, and for some it may be entirely ok to just flip the normals with `faceforward`.)

Here is the rib file for generating three irradiance cache files: wall.icf, floor.icf, and hero.icf. Note the settings of the various attributes.

```
FrameBegin 0

  Format 300 300 1
  ShadingInterpolation "smooth"
  PixelSamples 4 4
  Display "baking_indirdiff0" "it" "rgba"
  Projection "perspective" "fov" 30
  Translate 0 0 5

  WorldBegin

    Attribute "dice" "rasterorient" 0   # view-independent dicing !
    Attribute "visibility" "trace" 1   # make objects visible to rays

    # Bright green wall
    AttributeBegin
      Color [0 1.5 0]
      Surface "constant"
      Polygon "P" [ -1 1 -1  -1 1 1  -1 -1 1  -1 -1 -1 ]
    AttributeEnd

    # Wall
    AttributeBegin
      Surface "indirectsurf1sided" "samples" 1024
      Attribute "irradiance" "filemode" "w" "handle" "wall.icf"
      Polygon "P" [ -1 1 1  1 1 1  1 -1.1 1  -1 -1.1 1 ]
    AttributeEnd

    # Floor
    AttributeBegin
      Surface "indirectsurf1sided" "samples" 1024
      Attribute "irradiance" "filemode" "w" "handle" "floor.icf"
      Polygon "P" [ -0.999 -1 1  0.999 -1 1  0.999 -1 -0.999  -0.999 -1 -0.999]
    AttributeEnd

    # "Hero character": cylinder, cap, and sphere
    AttributeBegin
      Opacity [0.5 0.5 0.5]   # to ensure occlusion is computed behind objects
      Translate -0.3 -1 0   # this is the local coord sys of the "hero"
      CoordinateSystem "hero_coord_sys0"
      Surface "indirectsurf1sided" "coordsystem" "hero_coord_sys0" "samples" 1024
      Attribute "irradiance" "filemode" "w" "handle" "hero.icf"
      Attribute "irradiance" "maxerror" 0.1
      AttributeBegin
        Translate 0 0.3 0
        Scale 0.3 0.3 0.3
        ReadArchive "nurbscylinder.rib"
      AttributeEnd
      AttributeBegin
        Translate 0 0.6 0
        Rotate -90 1 0 0
        Disk 0 0.3 360
      AttributeEnd
      AttributeBegin
        Translate 0 0.9 0
        Sphere 0.3 -0.3 0.3 360
      AttributeEnd
    AttributeEnd

  WorldEnd
FrameEnd
```

Figure 3.20 shows the rendered image along with the three generated irradiance cache files.

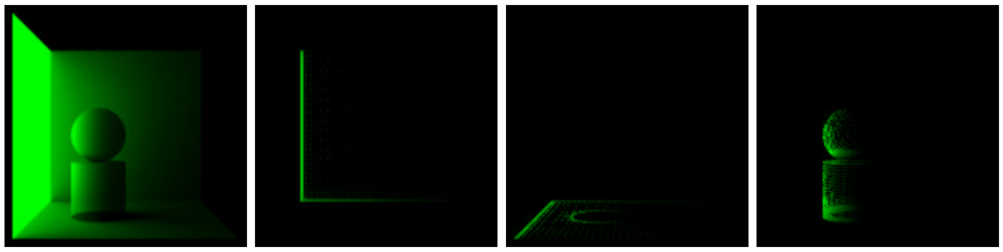


Figure 3.20: Irradiance cache file generation: (a) image; (b) wall.icf; (c) floor.icf; (d) hero.icf.

The images clearly illustrate that the irradiance cache values are placed adaptively: most irradiance values are placed where the irradiance changes a lot. The `"irradiance" "maxerror"` attribute was set to 0.5 (the default) for the wall and floor and to 0.1 for the hero. (If the `maxerror` attribute is set to 0, an irradiance value is computed at all shading points — this is very time consuming and generates an overwhelming amount of data unless the shading rate is set to a high value.)

As already mentioned, when the irradiance is computed, the ambient occlusion is computed as well since the additional computation is for free. So an irradiance cache file contains both irradiance and ambient occlusion values.

Next we want to reuse the baked irradiance values. Here is a shader that calls the `indirect-diffuse` function. Note that it also calls the `diffuse` function to compute direct illumination from light sources (of which there are none in this scene).

```
surface indirectsurf(float samples = 16, minsamples = 16, maxdist = 1e30;
                     string coordsystem = "", envmap = "")
{
  normal Ns = shadingnormal(N);

  Ci = diffuse(Ns) +
       indirectdiffuse(P, Ns, samples, "maxdist", maxdist,
                       "coordsystem", coordsystem,
                       "environmentmap", envmap);

  Ci *= Cs * Os;
  Oi = Os;
}
```

There is a similar shader using Pref points for the deformed cylinder.

The following is the scene description for rerendering the scene from a different camera viewpoint and with the "hero" moved away from the bright green patch. We reuse the baked irradiance on the wall and hero, but cannot reuse the irradiance on the floor.

```
FrameBegin 3

  Format 300 300 1
  PixelSamples 4 4
  ShadingInterpolation "smooth"
  Display "baking_indirdiff frame 3" "it" "rgba"
  Projection "perspective" "fov" 30
  Translate 0 0 5
  Rotate -20 1 0 0

  WorldBegin

    Attribute "visibility" "trace" 1
    Sides 1

    # Bright green wall
    AttributeBegin
      Color [0 1.5 0]
      Surface "constant"
      Polygon "P" [ -1 1 -1  -1 1 1  -1 -1 1  -1 -1 -1 ]
    AttributeEnd

    # Wall
    AttributeBegin
      Color [0 1 0]
      Surface "indirectsurf"
      Attribute "irradiance" "filemode" "R" "handle" "wall.icf"
      Polygon "P" [ -1 1 1  1 1 1  1 -1 1  -1 -1 1 ]
    AttributeEnd

    # Floor -- no baking reuse
    AttributeBegin
      Surface "indirectsurf" "samples" 1024
      Attribute "irradiance" "filemode" "" "handle" "floor.icf"
      Polygon "P" [ -1 -1 1  1 -1 1  1 -1 -1  -1 -1 -1 ]
    AttributeEnd

    # Cylinder + cap + sphere
    AttributeBegin
      Translate 0.3 -1 0
      CoordinateSystem "hero_coord_sys3"
      Surface "indirectsurf" "coordsystem" "hero_coord_sys3"
      Attribute "irradiance" "filemode" "R" "handle" "hero.icf"
      AttributeBegin
        Surface "indirectsurfPref" "coordsystem" "hero_coord_sys3"
        Translate 0 0.3 0
        Scale 0.3 0.3 0.3
        ReadArchive "nurbscylinder_deformed.rib"
      AttributeEnd
      AttributeBegin
        Translate 0 0.6 0
        Rotate -90 1 0 0
        Disk 0 0.3 360
      AttributeEnd
      AttributeBegin
        Translate 0 0.9 0
        Sphere 0.3 -0.3 0.3 360
      AttributeEnd
    AttributeEnd

  WorldEnd
FrameEnd
```

Note that the green irradiance on the "hero" is actually too bright and wraps around the cylinder and sphere too far — since he is now much further away from the bright green patch than when the irradiance was computed. It is a matter of judgment whether this is acceptable. A simple adjustment is to decrease the weight of the irradiance on the hero. The amount that the green illumination wraps around the cylinder and sphere would remain wrong, but that may be less objectionable than the incorrect brightness.

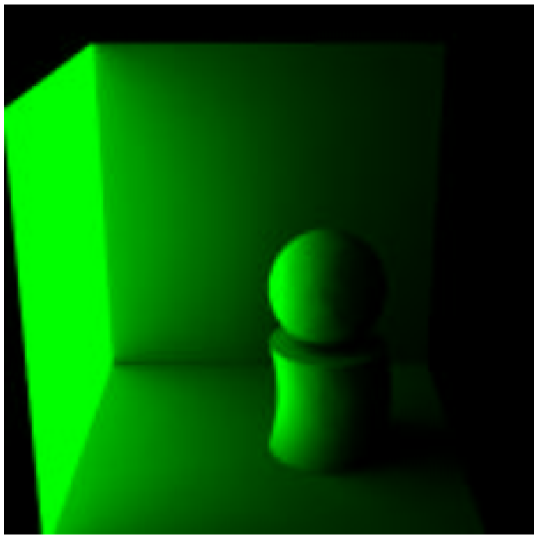


Figure 3.21: Reuse of baked irradiance.

3.8 Conclusion

We have presented an overview of how to render ambient occlusion, image-based illumination, and global illumination using PRMan. We have also presented methods to bake and reuse these quantities to amortize the cost of computing them over multiple frames.

It is our hope that the examples in this course note will contribute to "demystifying" global illumination and make it a more standard tool in the toolboxes of technical directors. We are committed to dispell the misconception that global illumination is still too inflexible, slow, and limited to consider for real production use.

Acknowledgements

Thanks to my colleagues in Pixar's RenderMan Products Group for numerous interesting discussions regarding ambient occlusion, image-based illumination, and global illumination, and for emphasizing the need for fast, flexible, and user-friendly global illumination methods in "real world" movie production work. Thanks to Tony Apodaca for helping us in the initial phase of the ray tracing project (explaining the mysteries of Opal), to Dylan Sisson for testing and great test scenes, to Mitch Prater for motivation, more great test scenes, and moral support, and to Mark VandeWettering for testing and many feature suggestions. Many thanks also to our terrific customers who use these features of RenderMan in fantastic and unexpected ways.

References

[1] Anthony A. Apodaca. Photosurrealism. In *Proceedings of the 9th Eurographics Workshop on Rendering*, pages 315–322. Springer-Verlag, 1998. (Also published as chapter 1 of Anthony A. Apodaca and Larry Gritz, *Advanced RenderMan — Creating CGI for Motion Pictures*, Morgan Kaufmann Publishers, 2000).

[2] Ronen Barzel. Lighting controls for computer cinematography. *Journal of Graphics Tools*, 2(1):1–20, 1997. (Also published as chapter 14 of Anthony A. Apodaca and Larry Gritz, *Advanced RenderMan — Creating CGI for Motion Pictures*, Morgan Kaufmann Publishers, 2000).

[3] Rob Bredow. RenderMan on film. In *SIGGRAPH 2002 course note #16*, pages 103–128. ACM, July 2002.

[4] Per H. Christensen, David M. Laur, Julian Fong, Wayne L. Wooten, and Dana Batali. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. *Computer Graphics Forum (Proceedings of Eurographics 2003)*, 22(3), 2003. (To appear).

[5] Cyrille Damez, Philipp Slusallek, Per H. Christensen, Karol Myszkowski, Ingo Wald, and Bruce J. Walter. Global illumination for interactive applications and high-quality animations. *SIGGRAPH 2003 course note #27*. ACM, July 2003.

[6] Paul E. Debevec. Rendering synthetic objects into real scenes: bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Computer Graphics (Proceedings of SIGGRAPH 98)*, pages 189–198, July 1998.

[7] Paul E. Debevec and Jitendra Malik. Recovering high dynamic range radiance maps from photographs. In *Computer Graphics (Proceedings of SIGGRAPH 97)*, pages 369–378, August 1997.

[8] Larry Gritz and James K. Hahn. BMRT: A global illumination implementation of the RenderMan standard. *Journal of Graphics Tools*, 1(3):29–47, 1996.

[9] Henrik Wann Jensen. *Realistic Image Synthesis using Photon Mapping.* A K Peters, 2001.

[10] Henrik Wann Jensen, Frank Suykens, Per H. Christensen, and Toshi Kato. A practical guide to global illumination using photon mapping. *SIGGRAPH 2002 course note #43*. ACM, July 2002.

[11] Hayden Landis. Production-ready global illumination. In *SIGGRAPH 2002 course note #16*, pages 87–102. ACM, July 2002.

[12] Philipp Slusallek, Thomas Pflaum, and Hans-Peter Seidel. Implementing RenderMan — practice, problems, and enhancements. In *Computer Graphics Forum (Proceedings of Eurographics '94)*, pages 443–454, 1994.

[13] Philipp Slusallek, Thomas Pflaum, and Hans-Peter Seidel. Using procedural RenderMan shaders for global illumination. In *Computer Graphics Forum (Proceedings of Eurographics '95)*, pages 311–324, 1995.

[14] Pixar Animation Studios. RenderMan Interface Specification version 3.2, July 2000. (https:-//renderman.pixar.com/products/rispec/index.htm).

[15] Pixar Animation Studios. Ambient occlusion, image-based illumination, and global illumination. *PhotoRealistic RenderMan Application Note #35*, 2002.

[16] Pixar Animation Studios. Caustics. *PhotoRealistic RenderMan Application Note #34*, 2002.

[17] Pixar Animation Studios. A tour of ray traced shading in PRMan. *PhotoRealistic RenderMan Application Note #36*, 2002.

[18] Gregory J. Ward and Paul Heckbert. Irradiance gradients. In *Proceedings of the 3rd Eurographics Workshop on Rendering*, pages 85–98. Eurographics, May 1992. (Also published in Greg Ward Larson and Rob Shakespeare, *Rendering with Radiance — The Art and Science of Lighting Visualization*, Morgan Kaufmann Publishers, 1998).

[19] Sergei Zhukov, Andrei Iones, and Gregorij Kronin. An ambient light illumination model. In *Rendering Techniques '98 (Proceedings of the 9th Eurographics Workshop on Rendering)*, pages 45–55. Springer-Verlag, 1998.

Chapter 4

Implementing a Skin BSSRDF (or Several...)

Christophe Hery
Industrial Light + Magic

Introduction

In the seminal paper from 2001: *A Practical Model for Subsurface Light Transport* ([Jensen '01]), Henrik Wann Jensen et al employed the concept of a BSSRDF (a bidirectional surface scattering distribution function) as a means to overcome the limitations of BRDFs.

The following year, in *A Rapid Hierarchical Rendering Technique for Translucent Materials* ([Jensen '02]), Jensen and Buhler presented a caching method to accelerate the computation of the BSSRDF.

We will try here to give pointers about how to implement such systems, as well as how to use them in production.

Finally, this section of the course can also be seen as an extension of Matt Pharr's presentation from 2001: *Layered Media for Surface Shaders* ([Pharr '01]).

4.1 BSSRDF 101

Skin is a multi-layered medium, in which photons tend to penetrate and scatter around many many times before bouncing back out. Certain wavelengths are attenuated differently according to the thickness. The path the photons take is highly complex and depends on the nature of the substrates as well as their energy. But one thing is almost certain: they rarely leave through the location where they entered, which is why a BSSRDF - a model for transport of light through the surface - is needed.

The full BSSRDF system presented by Jensen et al. consists of two terms: a single scattering component, which through path tracing gives us an approximate solution for the cases where light bounced only once inside the skin, and a multiple scattering component, through a statistical dipole point source diffusion approximation. The beauty of this model is that it can be implemented in a simple ray-tracer, and with the use of some trickery, in a Z-buffer renderer.

We are now going to go over the papers and extract the relevant information for shader writers, so please have your Proceedings handy.

4.1.1 Single Scattering

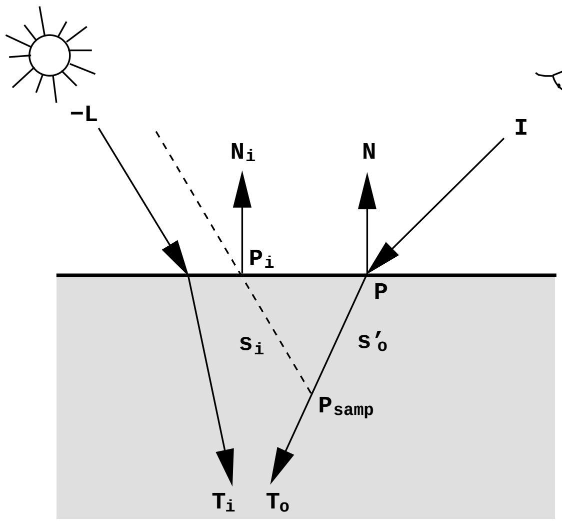


Figure 4.1: For single scattering, we march along the refracted outgoing ray T_o and project towards L.

We are trying to consider the different paths photons could take such that they would exit towards the camera at the shaded point after one unique bounce inside the skin. Since we know the camera viewpoint $\underline{\mathsf{I}}$ at the shaded surface position $\underline{\mathsf{P}}$, we can compute the refracted outgoing direction T_o. So let's select some samples P_{samp} along T_o (we'll figure out the density and the placement of those samples later), and let's try to find out how the light could have bounced back at those positions.

On the way in, the light should have refracted at the surface boundary. As mentioned in [Jensen '01], this is a hard problem for arbitrary geometry, and we make the approximation that the light did not refract. With this simplification, we can now find for each P_{samp} where the light first entered by simply tracing towards the light source.

In fact, we will not really trace towards the source position. Given the distances involved, we can consider that the light is uniform in direction and in color/intensity over the surface. So we simply trace along $\underline{\mathsf{L}}$ (the incident light direction at $\underline{\mathsf{P}}$) and multiply our result globally with $\underline{\mathsf{Cl}}$ (light energy received at $\underline{\mathsf{P}}$).

At that stage, we have: the distance the light traveled on the way in (s_i), and the distance the light traveled going out (s'_o). Prime signs here indicate, where appropriate, the refracted values. For instance, s'_o is the distance along the refracted outgoing ray, T_o, and it is a given, since we originally explicitly placed our sample P_{samp} along T_o from $\underline{\mathsf{P}}$ (in other words, $\mathsf{P}_{samp} = \underline{\mathsf{P}} + \mathsf{T}_o s'_o$).

The outscattered radiance for the single term, L_o, depends on the scattering coefficients, the phase function between the refracted directions, and some exponential falloff terms on s'_i and s'_o. But we do not know s'_i, the refracted distance before the bounce. The trick here is to use Snell's law to derive s'_i from s_i.

The other thing we are missing is the phase function. What is it? Simply stated, the phase function between two vectors $\vec{v}_1$ and $\vec{v}_2$ is a way to describe the amount of light scattered from the direction $\vec{v}_1$ into the direction $\vec{v}_2$. Knowing fully that we are not simulating, in CG, the light interactions one photon at a time, this phase becomes a probability density function (PDF), and for skin, it has been shown (for instance in [Pharr '01]) that the Henyey-Greenstein model is adequate.

Finally, we can try to make the solution converge faster. A naive implementation would have all P_{samp} uniformly selected along T_o. But this would require many many samples.

Let's look at L_0:

$$L_o(x_o, \vec{\omega}_o) = \frac{\sigma_s(x_o) F p(\vec{\omega}_i' \cdot \vec{\omega}_o')}{\sigma_{tc}} e^{-s_i' \sigma_t(x_i)} e^{-s_o' \sigma_t(x_o)} L_i(x_i, \vec{\omega}_i).$$

This expression can clearly be rewritten as the product of the exponential falloff in s_o' with another function:

$$L_o(x_o, \vec{\omega}_o) = [L_i(x_i, \vec{\omega}_i) \frac{\sigma_s(x_o) F p(\vec{\omega}_i' \cdot \vec{\omega}_o')}{\sigma_{tc}} e^{-s_i' \sigma_t(x_i)}] e^{-s_o' \sigma_t(x_o)}.$$

If we now pick the samples according to the PDF

$$x \sim \sigma_t e^{-\sigma_t x},$$

the integration in L_0 can be approximated by a simple summation. This is called importance sampling. By choosing

$$s_o' = \frac{-\log(random())}{\sigma_t},$$

we respect our PDF, and we can sum up all contributions without explicitly computing the falloff term in s_o'.

In pseudo shading language code, this becomes:

```
float singlescatter = 0;

vector To = normalize(refract(I,N,oneovereta));

for (i=0; i<nbrsamp; i+=1)
{
  float sp_o = -log(random())/sigma_t;
  point Psamp = P + To * sp_o;
  point (Pi,Ni) = trace(Psamp, L);
  float si = length(Psamp - Pi);
  float LdotNi = L.Ni;
  float sp_i = si * LdotNi
      / sqrt (1 - oneovereta*oneovereta * (1 - LdotNi*LdotNi));
  vector Ri, Ti; float Kri, Kti;
  fresnel (-L, Ni, oneovereta, Kri, Kti, Ri, Ti);
  Kti = 1-Kri;
  Ti  = normalize(Ti);
  float g2 = g*g;
```

```
    float phase = (1-g2) / pow(1+2*g*Ti.To+g2,1.5);
    singlescatter += exp(-sp_i*sigma_t) / sigma_tc * phase * Kti;
  }

  singlescatter *= Cl * PI * sigma_s / nbrsamp;
```

There is also a fresnel outgoing factor, but since it is to be applied for both the single and the diffusion terms, we are omitting it in this section.

It is that simple (in fact it is made simpler here, because we only deal with one wavelength)!

Lastly, note the trace call. It returns the position, the normal and potentially the textured diffuse color of the intersection of the ray going from P_{samp} in the $\underline{\mathsf{L}}$ direction. We will come back to it in section 4.3.

4.1.2 Diffusion Scattering

We are trying to accumulate the observed profile of illumination over all positions on the surface. The dipole of point sources is a virtual construction to simulate the light attenuation on skin reported by medical scientists (the derivation of this model is beyond the scope of our discussion). The value we are summing up, at each sample, is the diffuse re¤ectance:

$$R_d(r) = \frac{\alpha'}{4\pi}[(\sigma_{tr}d_r+1)\frac{e^{-\sigma_{tr}d_r}}{\sigma_t' d_r^3} + (\sigma_{tr}d_v+1)\frac{e^{-\sigma_{tr}d_v}}{\sigma_t' d_v^3}]$$

Its expression mainly depends on the distance r from the shaded point to the sample, and also (directly and indirectly) on the scattering coefficients. The problem is once again the distribution of those samples. In fact, in [Jensen '02], a two-pass pre-distribution technique was showed (and we will come back to it in section 4.4). For now, let's assume that we want to place those samples at their appropriate locations during shading time. They need to lie on the surface and be located around $\underline{\mathsf{P}}$. The $\sigma_{tr}e^{-\sigma_{tr}d}$ terms are likely going to dominate the computation of R_d (both in times and in values). So we should try to put in a lot of effort in order to make them converge faster. Again we turn to the importance sampling theory, and we choose a PDF

$$x \sim \sigma_{tr}^2 e^{-\sigma_{tr}x}.$$

Why do we have a σ_{tr}^2 coefficient this time (where our PDF for single scattering had a simple σ_t multiplicator)? Well, we need to integrate in polar coordinates (whereas for single scattering we were marching along T_o in 1D), and a condition for a valid PDF is that its integral over the full domain comes to 1 (meaning that we have 100% chance to find a sample in the infinite range). This PDF corresponds to the cumulative density function (CDF)

$$P(R) = \int_0^R \sigma_{tr}^2 e^{-\sigma_{tr}r} r dr = 1 - e^{-\sigma_{tr}R}(1+\sigma_{tr}R)$$

By the change of variable $u = \sigma_{tr}R$, we get

$$P(u) = 1 - e^{-u}(1+u),$$

and through a numerical inversion of $P(u)$, we get a table of precomputed positions, that we can insert in the shader (we provide a C program in the Appendix that warps a uniform Poisson distribution into the one we want).

Again let's deal with the pseudo-code for one wavelength.

```
float diffusionscatter = 0;

uniform float scattdiffdist1[maxSamples] = { /* X table */ };
uniform float scattdiffdist2[maxSamples] = { /* Y table */ };

/* create a local frame of reference for the distribution */
vector  local = N;
vector  up    = vector "world"(0,1,0);
if (abs(local.up) > 0.9)
    up    = vector "world"(1,0,0);
vector  base1 = normalize(local^up);
vector  base2 = local^base1;

for (i=0; i<nbrsamp; i+=1)
{
  point Psamp = P + 1/sigma_tr *
      (base1 * scattdiffdist1[i] + base2 * scattdiffdist2[i]);
  point (Pi,Ni) = trace(Psamp, -local);
      /* make sure we are on the surface */

  float r = distance(P, Pi);
  float dr = sqrt(r*r+zr*zr); /* distance to positive light */
  float dv = sqrt(r*r+zv*zv); /* distance to negative light */
  float sigma_tr_dr = sigma_tr*dr;
  float sigma_tr_dv = sigma_tr*dv;
  float Rd = (sigma_tr_dr+1.0) * exp(-sigma_tr_dr) * zr/(dr^3)
          + (sigma_tr_dv+1.0) * exp(-sigma_tr_dv) * zv/(dv^3);

  scattdiff += Rd / (sigma_tr*sigma_tr * exp(-sigma_tr*r));
      /* importance sampling weighting */
}

scattdiff *= Cl * L.N * (1-Fdr) * alpha_prime / nbrsamp;
```

Please be aware that, in this example, the distribution was done on the tangent plane at $\underline{P}$. Depending on the chosen tracing approach to reproject the samples on the actual geometry, this might not be the best solution.

4.2 Texture Mapping and Re-Parameterization

So far we have considered a shader that depends on some scattering coefficients. In reality, we need to texture them over our geometry. Painting them directly is non-trivial, and especially non-intuitive. In production, we tend to start from photographs of our live characters or maquettes. The question is: can we extract the variation of the relevant parameters from regular diffuse texture maps?

We are going to make some simplifications to make this conversion. First, we will assume that the in¤uence of the single scattering is negligible compared to the diffusion term ([Jensen ’02] provides a justification for skin materials). So let’s assume that our diffuse texture map is the result of the diffusion scattering events under a uniform incident illumination, and try to solve for the scattering parameters which produce these colors.

In this case, we can approximate our BSSRDF as a BRDF:

$$R_d = \frac{\alpha'}{2}(1+e^{-\frac{4}{3}A\sqrt{3(1-\alpha')}})e^{-\sqrt{3(1-\alpha')}},$$

which only depends on α' and A (indirectly η). This is also equal, by construction, to our diffuse map $\mathsf{C}_{\mathsf{map}}$. So we now have one equation with two unknowns: α' and η. If we fix η (at 1.3 for skin), we can get α' as a function of $\mathsf{C}_{\mathsf{map}}(= R_d)$. In truth, we cannot do this analytically, but we can build yet another table numerically.

We used the following code in Matlab to produce the table:

```
clear;
eta = 1.3;
Fdr = -1.440/(eta.^2) + 0.710/eta + 0.668 + 0.0636*eta;
A = (1 + Fdr)/(1 - Fdr);
alpha = 0:.01:1;
c = alpha.*(1+exp(-4/3*A*sqrt(3*(1-alpha))))
         .* exp(-sqrt(3*(1-alpha)));
C = 0:.001:2;
ALPHA = interp1(c,alpha,C);
```

And, in the shader, we can lookup those values in the following manner:

```
uniform float alpha_1_3[2001] = { /* put matlab table here */ };
float alpha_prime = alpha_1_3[floor(diffcolor*2000.0)];
```

Because α' is the reduced transport albedo, namely the ratio:

$$\alpha' = \frac{\sigma_s'}{\sigma_a + \sigma_s'},$$

this does not resolve directly into the scattering coefficients.

At ILM, we have been using in the last couple of years the idea of fixing the σ_s' parameter, and deducing σ_a from the maps and the α table:

$$\sigma_a = \sigma_s' \frac{1-\alpha'}{\alpha'}$$

Even though σ_s' is given for some specific materials in [Jensen ’01], it is still a very non-intuitive parameter, hence at ILM we have begun to replace our tuning factor with the diffuse mean free path $ld =$

$1/\sigma_{tr}$. It turns out that σ_{tr} and α' are all that is needed to control the diffusion scattering. For the single scattering, we can get:

$$\sigma_t' = \frac{1}{ld\sqrt{3(1-\alpha')}}$$

and

$$\sigma_s' = \alpha' \sigma_t'.$$

Lastly, it is worth mentioning that, ideally, the color inversion should be done at every sample for diffusion scattering, so that we get the correct R_d on those positions. For single scattering, the dependence on the inversion during the ray marching is embedded in the σ_{tc} denominator and σ_t term in the exponential. So again, ideally, we should run the inversion for every sample.

4.3 BSSRDF Through Z-Buffers

If we can manage to *trace* only along the light direction, we can achieve a projection towards $\underline{\mathsf{L}}$ by some simple operations on the shadow buffers, of the kind:

```
uniform matrix shadCamSpace, shadNdcSpace;
textureinfo(shadMap, "viewingmatrix", shadCamSpace);
textureinfo(shadMap, "projectionmatrix", shadNdcSpace);
uniform matrix shadInvSpace = 1/shadCamSpace;
point Pi       = transform(shadCamSpace, Psamp);
point tmpPsamp = transform(shadNdcSpace, Psamp);
float shadNdcS = (1.0+xcomp(tmpPsamp)) * 0.5;
float shadNdcT = (1.0-ycomp(tmpPsamp)) * 0.5;
float zmap = texture(shadMap,
                     shadNdcS, shadNdcT, shadNdcS, shadNdcT,
                     shadNdcS, shadNdcT, shadNdcS, shadNdcT,
                     "samples", 1);
if (comp(shadNdcSpace, 3, 3) == 1) /* orthographic */
    setzcomp(Pi, zmap);
else
    Pi *= zmap/zcomp(Pi);
Pi = transform(shadInvSpace, Pi);
```

To access any other values at P_i (like N_i), if we produced an image of those values from the point of view of the light (for instance through a secondary display output), we simply lookup the texture, as in:

```
color CNi = texture(shadCNMap,
                    shadNdcS, shadNdcT, shadNdcS, shadNdcT,
                    shadNdcS, shadNdcT, shadNdcS, shadNdcT,
                    "samples", 1);
normal Ni;
setxcomp(Ni, comp(CNi,0));
```

```
setycomp(Ni, comp(CNi,1));
setzcomp(Ni, comp(CNi,2));
Ni = ntransform(shadInvSpace, Ni);
```

Of course, if we want to emulate the soft shadows cast from a non-point light source, we can supersample our shadNdcS and shadNdcT lookup coordinates or any relevant tracing value in the shader.

This overall tracing method drops right in for single scattering (the trace is indeed along $\underline{\mathsf{L}}$), but it is more complex to make it work for diffusion scattering.

Let's think about it for a minute. We need to place some samples on the surface according to a specific distribution. Our assumption was that we would first position them on the tangent plane at $\underline{\mathsf{P}}$, then project them (along $-\underline{\mathsf{N}}$) to the surface. With our Z-buffers, all we have is light projections. So what if we distributed our samples in shadow space, then projected along $-\underline{\mathsf{L}}$? The surface distribution would definitely be skewed. Our importance sampling mechanism would not converge as fast as expected (it would still be faster than a simple uniform distribution though). In fact, by doing it this way, it is almost as if we would be building implicitly the $\underline{\mathsf{L}} \cdot \underline{\mathsf{N}}$ product. We should then just "correct" the final result by omitting this factor.

In our pseudo-code from section 4.3, we just change:

- `vector local = N` into `vector local = L`,
- the trace with the method exposed above,
- the last line into `scattdiff *= Cl * alpha_prime / nbrsamp`

In practice, this twist on the theory works really well.

Ideally, we need 3 distributions (one per wavelength), ie 3 sets of fake trace Z-buffer projections. Not only is it a slow process (after all, `texture` is one of the most expensive operators in the shading language), but doing it that way creates a separation of the color channels, hence a very distracting color grain. The remedy is to do only one distribution with the minimum σ_{tr} (the wavelength that scatters the most).

Finally, since we use the shadow buffers as a way to represent the surface of our skin, we should not include in them any other foreign geometries, like clothing or anything casting shadows from an environment (more on that later at paragraph 4.5.1). For the same reason, the resolution of the Z-buffers matters a great deal.

4.4 BSSRDF Through a Point Cloud Cache

As we said before (in section 4.1.2), the diffusion scattering term largely dominates the BSSRDF in the case of human skin rendering, to the point where we should think about dropping single scatter altogether from our model of skin. Single scatter can be made to produce good results though. One can balance it, tweak it, blur it, contain it with some area maps, so that its appearance is pleasing. However, from its very nature of being a directional term, the transitions of the illumination (especially the transitions from back lighting to front lighting) are very hard. For other materials with more isotropic scattering properties (scattering eccentricity closer to 0), such as marble, it becomes more critical to keep this term. In any case, when we speak about a skin BSSRDF from this section on, we will only consider the diffusion scattering.

Since the BSSRDF computation is fairly expensive, it is worth looking at acceleration methods. One observation we can make is that, no matter the pattern of distributions of the dipole samples, they are likely going to be extremely close from one shaded point to the next, yet we do indeed recompute their positions for each $\underline{P}$. So one idea here would be to pre-distribute the samples on the surface, this time uniformly, then "gather" them during the scattering loop. This is what [Jensen '02] demonstrates.

There are several approaches one can take for this pre-distribution phase. One can build fancy Xsi scripts or mel commands that could achieve, through particle repulsion algorithms, a stabilized semi-Poisson set on the surface. One can implement a stand-alone Turk method ([Turk 1992]) as suggested in [Jensen '02]. Or one can take advantage of a specific rendering engine. For example, in the case of Pixar's PhotoRealistic RenderMan, one can derive the positions directly from the micro-polygon grids through a dso (a la Matt Pharr's `http://graphics.stanford.edu/~mmp/code/dumpgrids.c`), or maybe, more directly, Pixar will provide some hooks like extending the new micro-polygon caching hider. Finally, one can also use the fancy irradiance caching mechanism: this is indeed exactly the type of information needed to be stored in our cache. All in all, there are certainly many more ways to obtain a similar point cloud representation of the geometry.

So let's assume that we managed to pre-generate this distribution, and that it is stored in a file for which we know the format. The relevant information for each sample should be its location, the normal, the coverage area it represents - in other words its local sampling density (this is necessary in the cases for which our distribution is not exactly uniform) - and, if possible, the diffuse texture at that position.

The first thing to do will be to compute the diffuse irradiance at each point, and store it back in the data structure. Again there are several ways to do this (in fact, if we opted to do the distribution through Matt Pharr's method or through the irradiance cache, the value is already present in the file, and we can step over this stage).

One hacky approach is to read the file one sample at a time through a shader DSO, then do an illuminance loop on the position and the normal just obtained, compute the $\underline{L} \cdot N_{cache}$ factor, and write the whole thing back. This is the code for such a scheme:

```
surface lightbake_srf
  (
  string inputScattCache  = "";
  string outputScattCache = "";
  float doFlipNormals = 0.0;
  )
{
   if (opencache(inputScattCache) == 1) {
      float index = indexcache();
      while(index >= 0.0) {
          point  cacheP = getPcache(index);
          normal cacheN = getNcache(index);
          cacheP = transform("world", "current", cacheP);
          if (doFlipNormals == 1)
              cacheN = ntransform("world", "current", -cacheN);
          else
              cacheN = ntransform("world", "current", cacheN);
          illuminance(cacheP, cacheN, PI/2) {
              float LdotN = L.cacheN;
              if (LdotN > 0.0) {
                  accumcache(index, Cl * LdotN / length(L));
              }
          }
          index = indexcache();
      }
      writecache(outputScattCache);
   }
}
```

This is a pretty weird surface shader indeed: apart from the illuminance statement, it uses only DSO calls to access our file format. The cache is opened and kept in as a static member of the DSO at the opencache stage, so that we can look up any specific position quickly. We are using an index as a kind of a pointer to the current sample, so all information about it can be read and written correctly.

A word of caution here: the light shaders need to be modified to do their own point sampling on the shadows; otherwise, the derivatives, as far as RenderMan is concerned, will come from the successive cacheP values (which can jump all over the place, depending on how they are stored). Here is another trick. Instead of assigning the lightbake shader to a random visible piece of geometry, we use an RiPoint primitive, which has no derivatives by construction. The shadow calls are thus forced to point sample the buffers, without any editing of the lights code. Another side benefit is that the overall computation is much faster this way.

To obtain the final beauty BSSRDF result, in yet another DSO, we read our cache file and store the data in memory as an octree for quick lookup. Given a search radius R, we can sum up the R_d contributions from each sample found in the proximity region around $\underline{P}$. This is the same R_d computation as presented in section 1-b:

```
float Rd = (sigma_tr_dr+1.0) * exp(-sigma_tr_dr) * zr/(dr^3)
         + (sigma_tr_dv+1.0) * exp(-sigma_tr_dv) * zv/(dv^3);
```

But we want to pre-multiply it by the correct precomputed radiance at each sample and its coverage area. Also, if we have the color information in the file, we can re-derive σ_{tr}, zr and zv through the alpha inversion as mentioned in paragraph 4.2. Note that the surface shader does not have to do any queries of the lights or any *tracing* anymore; it just needs to call the DSO.

This scatter gathering stage can still be fairly slow, for the cache might contain several thousands samples (to accurately represent the geometry and local details in the illumination and textures), so we use a level-of-detail approach. We generate some cached distribution at several densities. We search locally in the finer cache, and then, as the exponential falloffs in R_d tend towards 0, we look up in the more sparse files. This way, we can minimize the number of returned sample positions. [Jensen '02] describes a more formal data structure for this kind of optimization.

4.5 Extensions

4.5.1 Ambient Occlusion, Global Illumination and Cast Shadows

Obviously the Z-buffer approach requires some sort of direct illumination, with some actual light sources. At first this makes it seem impossible to get an ambient occlusion (see last year's notes) contribution to participate in the scattering. Even though one can still do the ambient as a separate term in the shader, there is a sneaky way to merge the two together. The idea is to compute the occlusion, not through a shadeop or a ray-tracing pre-pass, but with a dome of lights. This method has recently been described on the web at: `http://www.andrew-whitehurst.net/amb_occlude.html`, so we are not going to cover it here.

Similarly, the skin position is assumed to be at the Z-buffer locations, so it does not seem easy to get correct cast shadows on our subject (we would not want to make the algorithm think that our surface is feet away from where it really is, right?). The solution to that is to generate a separate ”outer” shadow buffer for each light, and scale the final result based on it.

On the other hand, there is no such limitation on the point cloud method of computing the scattering. Because we light the samples as a pre-pass, we can use any type of illumination algorithms on them, even any type of rendering engine, ahead of accumulating the dipoles in the final render.

4.5.2 Concept of Blocking Geometries

Up to this point the shader hasn’t solved a small structural problem. It doesn’t take into account the fact that inside the skin there are bones that are basically blocking some of the scattering effect. This extension is critical for small creatures and/or highly translucent materials, where one wants to give the impression of an obscuring entity inside the medium.

With the Z-buffer approaches, this is pretty trivial. We just need to generate yet another shadow buffer from the point of view of the scattering lights, a buffer of those inner blocking geometries. Then we can point sample it during the loops to get a z_{inner} value. By comparing our projected z_{map} and original sample point zcomp(P_i) with this z_{inner}, we can decide if and how much the sample participates in the illumination. This method is valid for both single and diffusion scattering.

If our BSSRDF is based on the point cloud approach, we can somehow simulate this effect by distributing samples on the inner geometries, summing the diffusion scattering they generate as seen from our skin shading point P, then subtracting this in¤uence (or part of it) from our real surface diffusion scattering computation. This trick is certainly non-physical, but it seems to produce a pleasing effect.

4.6 Summary

We presented a set of techniques for implementing a production ready sub-surface scattering model in the shading language.

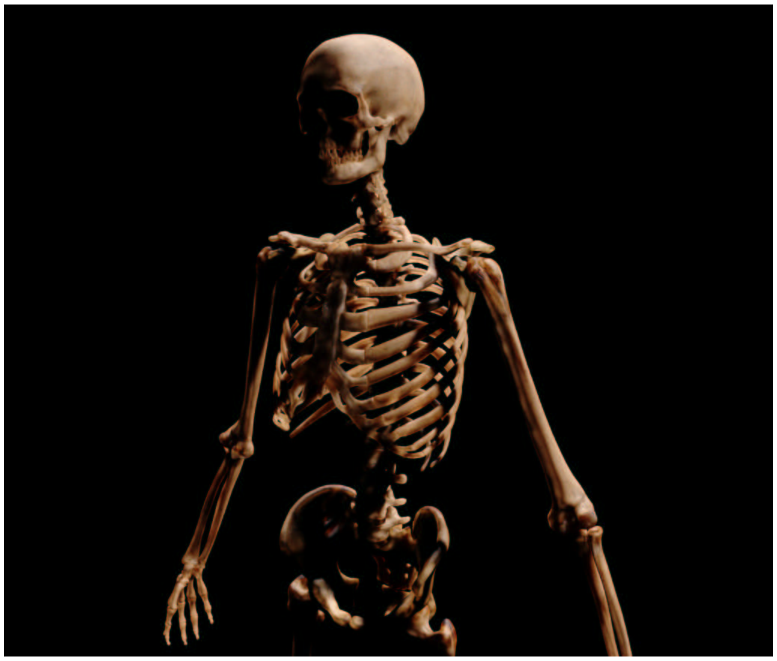


Figure 4.2: This skeleton was rendered with subsurface scattering using the point cloud approach and material properties similar to skin. One million sample positions were created in the high res distribution.

Acknowledgments

We would like to thank the following persons who all contributed one way or another in turning the theory into a practical solution. Dan B Goldman, Andrea Maiolo and Hilmar Koch for their shading savviness, Tony Pelle for his ”pointscatter” tool and its viewer, Phil Peterson for the LaTeXcrash course, and the entire ILM R&D department for supporting this effort and letting us publish it. These notes copyright (c) 2003, Industrial Light + Magic, a division of Lucasfilm. All rights reserved.

Appendix A

mktables.c

```
// mktables.c:
//   utility to generate a table corresponding to the distribution:
//   1 - exp(-r) * (1+r)

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

static const int numSampleTables = 1;
static const int maxSamples = 128;
static float sampleX[numSampleTables][maxSamples];
static float sampleY[numSampleTables][maxSamples];

static void randomPoint (float &u, float &v)
{
    do {
        u = drand48() - 0.5;
        v = drand48() - 0.5;
    } while (u * u + v * v > 0.25);
}

// First use the best candidate algorithm from [Mitchell 1991]:
// Spectrally Optimal Sampling for Distribution Ray Tracing
// Don P. Mitchell, Siggraph Proceedings 1991

void initSampleTables()
{
    const int q = 10;

    for (int table = 0; table < numSampleTables; table++) {
        randomPoint (sampleX[table][0], sampleY[table][0]);

        for (int i = 1; i < maxSamples; i++) {
            float dmax = -1.0;
            for (int c = 0; c < i * q; c++) {
                float u, v;
                randomPoint (u, v);

                float dc = 2.0;
                for (int j = 0; j < i; j++) {
                    float dj =
                        (sampleX[table][j] - u) * (sampleX[table][j] - u) +
                        (sampleY[table][j] - v) * (sampleY[table][j] - v);
                    if (dc > dj)
                        dc = dj;
                }

                if (dc > dmax) {
                    sampleX[table][i] = u;
                    sampleY[table][i] = v;
                    dmax = dc;
                }
            }
        }
    }
    return;
}

// Now take the uniform distribution and warp it into ours

inline float distFunc(float r)
{
    return(1.0-exp(-r)*(1.0+r));
}

float invDistFunc(float y, float tolerance)
{
    float test, diff;
    float x = y;
    while(1) {
        test = distFunc(x);
        diff = y - test;
        if (fabsf(diff) < tolerance) break;
        x = x + diff;
    }
    return(x);
```

```
}

void adaptSampleTables()
{
    float PI = fasin(1.0)*2.0;
    for (int i = 0; i < numSampleTables; i++) {
        for (int j = 0; j < maxSamples; j++) {
            float X = 2.0*sampleX[i][j]; // between -1 and 1
            float Y = 2.0*sampleY[i][j];
            float R = fsqrt(X*X+Y*Y);    // between 0 and 1
            float r = invDistFunc(R,.00001);
            float theta = fatan2(Y,X);   // between -PI and PI
            sampleX[i][j] = fcos(theta)*r;
            sampleY[i][j] = fsin(theta)*r;
        }
    }
}

// Finaly print the resulting tables

void printSampleTables()
{
    int i, j;

    printf ("uniform float scattdiffdist1[maxSamples] = {\n");
    for (i = 0; i < numSampleTables; i++) {
        for (j = 0; j < maxSamples; j++) {
            printf ("%f", sampleX[i][j]);
            if (j < maxSamples - 1)
                printf (",");
        }
        if (i < numSampleTables - 1)
            printf (",\n");
    }
    printf ("\n};\n");

    printf ("uniform float scattdiffdist2[maxSamples] = {\n");
    for (i = 0; i < numSampleTables; i++) {
        for (j = 0; j < maxSamples; j++) {
            printf ("%f", sampleY[i][j]);
            if (j < maxSamples - 1)
                printf (",");
        }
        if (i < numSampleTables - 1)
            printf (",\n");
    }
    printf ("\n};\n");
}

void main()
{
    initSampleTables();
    adaptSampleTables();
    printSampleTables();
    exit (0);
}
```

Chapter 5

Human Skin for “Finding Nemo”

Practical Subsurface Scattering with RenderMan®

Erin Tomson
Pixar
ekt@pixar.com

Rendering skin has become the canonical use of subsurface illumination techniques for good reason. Not only is skin quite translucent, but people are particularly perceptive when looking at human characters. Even if the character design doesn’t call for strict realism, as is the case with the humans characters in “Finding Nemo”, people still quickly get a sense that “something is wrong” when looking at skin rendered with a traditional BRDF model. Psychologically the impact is much greater than with other similarly translucent materials that lack subsurface illumination.

The humans in this film are not primary characters, yet it was clear that their skin still needed subsurface illumination. A system was needed that would provide this and meet the following criteria.

- Rendering should remain fast. The time for rendering skin should be negligible in comparison to the rendering time for hair.
- The system should be well integrated with existing lighting tools. The lighter should be able to enable scattering without changes to the lighting setup.
- Time was of the essence. A working system needed to be in place and ready for shading development very quickly.

In these course notes I describe the techniques I came up with for accomplishing this and give practical details for implementing it in your own RenderMan-based pipeline. I’ll describe the steps in a top to bottom fashion, starting with the mathematical scattering model and then working down from the final scattering computation to the initial point cloud generation technique.

5.1 Modeling the Scattering Computation

The skin rendering system for “Finding Nemo” used a simplified model of subsurface light transport. The subsurface illumination at a point P with normal N is the sum of the contributions of all the irradiance samples at other points on the surface.

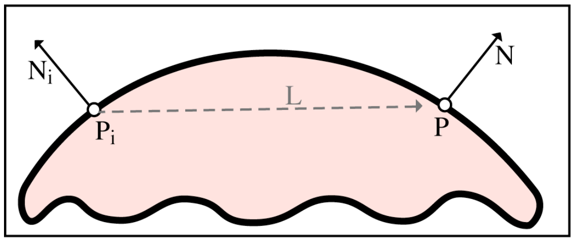
$$subsurf(P,N) = \sum A_i I(P_i,N_i) T(P,P_i) B(P,P_i,N,N_i)$$

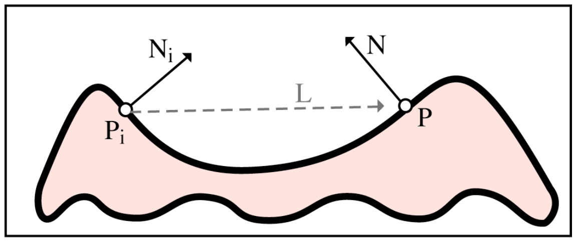
A_i is the area represented by sample i, $I(P_i,N_i)$ is the irradiance per unit area (the diffuse illumination at P_i), and $T(P,P_i)$ is the attenuation through the material from P_i to P.

$T(P,P_i)$ represents the amount of light that should be transferred between the sample point P_i and the point P that the scattering is being calculated for. I found that the good old *smoothstep*[1] function works remarkably well for this, once normalized to maintain brightness irrespective of the scattering distance, D.[2] This function looks similar to a gaussian but falls to zero at a finite distance.

$$T(P,P_i) = (1\text{-}smoothstep(0,D,length(P\text{-}P_i)))/(3D^2\pi/10)$$

So far the model assumes that the path between P and P_i is through skin, not air. This will generally be the case, or at least close to the case anytime P and P_i are on a surface that is locally flat or convex. But if the surface is locally concave (or worse, completely disjoint) then light will appear to ‘bounce’ from one surface to the other. Consider the following two cases.





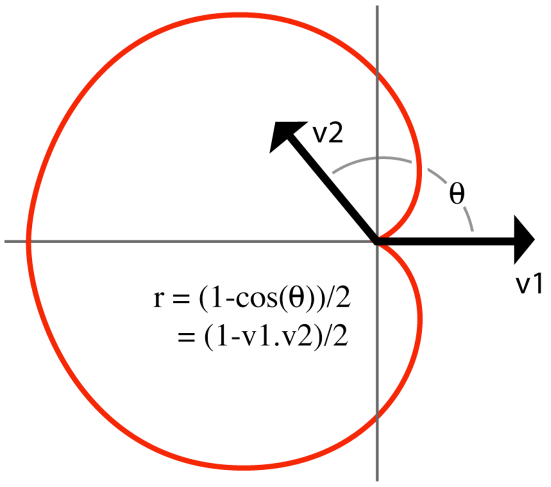
In the first case the path from P_i to P passes through skin. But in the second case the surface is locally concave. As a result, light will be transported through air and will appear to “bounce” from P_i to P. This undesirable effect can be overcome with the ‘bounce attenuation’ factor, $B(P,P_i,N,N_i)$.

The bounce problem occurs when the surface at the point being shaded faces the surface at the sample point. That is,

- the surface normal (*N*) and sample normal (N_i) oppose each other
- the surface normal (*N*) and the vector from the sample point to the surface point (*L*) oppose eachother.

[1] smoothstep(min,max,x) is defined as 0 if x is less than min, 1 if x is greater than max, or $-2y^3 + 3y^2$ otherwise, where y = (x-min)/(max-min).
[2] The integral of 1-smoothstep(0,D,sqrt(x^2+y^2)) over the plane is $3D^2\pi/10$.

After some experimentation, I found that a cardioid function for each of the two conditions effectively suppresses this problem with minimal artifacts.



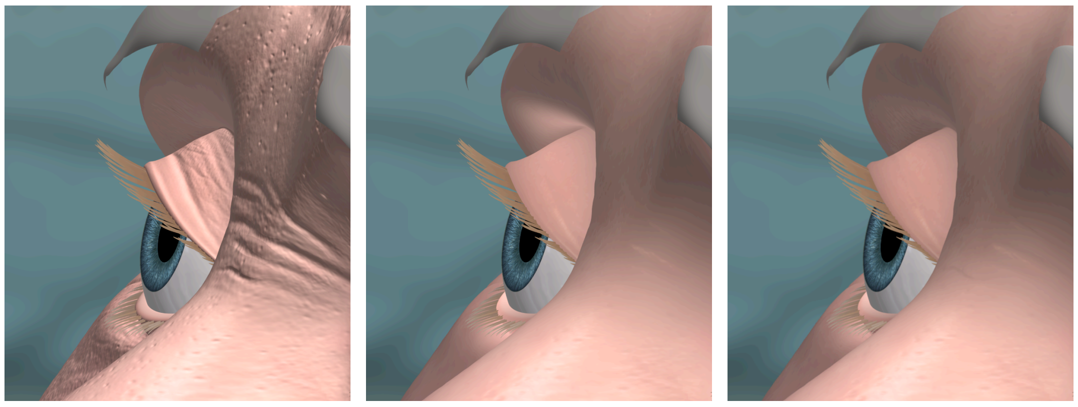
We wish to attenuate when

- The surface normal and sample normal oppose eachother: (1-N•N_i)/2 is near 1
- The surface normal and light vector oppose eachother[3]: (1-N•L) is near 1

So we define the bounce attenuation as[4]

$$B(P,N,P_i,N_i) \quad = 1\text{-}(1 - N.Ni)(1 - N.L)/2$$

The effects of the bounce attenuation factor are easy to see in the images below. The first image is rendered with no scattering. The second is with scattering but without bounce attenuation. Notice the bright spot below the character's brow. The third is the final rendering with bounce attenuation enabled.



[3] This term is twice the cardioid function since we wish it to approach 1 when as |L| approaches 0.

[4] The two normals, N and N_i are normalized to unit length. The vector L, however, is normalized by D and is therefore less than or equal to unit length. This avoids the instability that would otherwise arise from normalizing a very short vector.

5.2 Calculating the Scattering

Once a point cloud of irradiance samples (along with their associated points, normals, and areas) has been computed, the actual scattering computation is fairly straightforward. The steps are as follows.

- Build an octree out of the irradiance samples
- When shading point P, travere the octree to find all samples that have a distance from P less than or equal to D.
- Accumulate the subsurface illumination.

This is implemented as a C++ RenderMan DSO. The DSO reads the scatter map (a set of tiff files, described in section 4), calculates the scatter map's bounding box, and then recursively constructs an octree. Irradiance values are stored only at leaf nodes. When the number of samples in the leaf node exceeds a predefined maximum, the node is subdivided and the samples are moved to the node's new children.

Then when the shader runs on a point P, the point and it's normal are passed to the DSO. The DSO traverses the octree to find all the samples within a distance of D from P.

At each non-root node a child is traversed only if P is inside the child's bounding box or if the distance from P to the child is less than D. Finally when a leaf node is reached, the contribution from each sample in that leaf node is accumulated using the equation for subsurface illumination described in section 1. The accumulated value is returned to the shader and added to the other illumination components.

5.3 On-the-fly Irradiance Cache

In order to be integrated well with lighting tools it's helpful to calculate irradiance on the fly, rather than in a pre-pass. Since the number of irradiance samples in the scatter map is far fewer than the number of grid points being shaded, the computational expense of this is minimal. The benefit, however, is huge. It allows a lighter to change a lighting parameter and re-render without first re-rendering the scatter map. In fact, the scatter map only needs to be regenerated if the geometry or articulation changes.

5.3.1 Grid Counting

To accomplish this, the scattering DSO loads the points, normals, and areas from the scatter map into an octree. When the irradiance from a sample is first needed, the point, normal, and area of that sample are passed back to be shaded. The shader then calculates the irradiance and returns it to the DSO which caches it to speed future accesses. After all the samples that are needed have been illuminated in this manner, the shader calls the DSO again to compute the subsurface illumination.

In order to make all of this work, several RenderMan specific tricks must be employed. RenderMan shades in parallel, executing each statement on all of the points in a grid before executing the next statement on any.[5] Because of this parallel processing, a simple serial loop that reads a sample, illuminates it, and returns the value will not work.

This parallelism is addressed with a queue in the DSO. As samples are read from the DSO, they are added to the tail of the queue. Then as irradiance values are returned, they are stored with samples popped from the queue's head.

So if a grid has 220 samples then one line in the shader will cause 220 points, normals, and area values to be added to the queue and passed back to the shader. Another line in the shader will illuminate all 220 samples, and a third line will return all 220 illumination values to the DSO.

It important to fill the grid as much as possible when doing this. If the grid has 220 points and only 3 samples are irradiated, the light shaders will still be run 220 times! A "grid counting" mechanism helps avoid this inefficiency.

The first call to the DSO, 'ReadMap' passes in the name of the scattermap to use and resets an internal counter to 0. The second call, 'AddGridPoint' returns the current value of the counter and increments it. So, after 'AddGridPoint' has been been called, the counter contains the number of points in the grid and the return value contains a unique index for each grid point.

'AddGridPoint' also does the octree traversal, finding all the samples near each grid point. Any sample that hasn't been illuminated yet is added to a second queue of "required" samples. The irradiation loop will continue as long as there are more required samples or the grid has not yet been filled. That is, when the required samples queue is empty, samples from elsewhere in the octree will be returned until the grid is full.

At this point, all of the required samples have irradiance values and the octree has been traversed. A single final call accumulates the irradiance values and returns the final subsurface contribution.

5.3.2 Other Considerations

There are a couple other things to watch out for in this process. The first is the derivatives used for such things as filter widths of shadowmap lookups.

Since the grid being illuminated contains points that have no spatial coherence, the automatically computed derivatives must somehow be overridden. Message passing can be used to accomplish this. The actual micropolygon area of the sample is stored in the scattermap and then passed to the light shader for use in shadowmap accesses.

Another consideration is support for Irma, the re-renderer. Irma caches the results of all computations before the start of the shader's first lighting operation. Because of this, some sort of standard illumination function should be called before computing the subsurface scattering.

[5] This is the case except with statements that have only parameters and return values declared as uniform. Those are executed just once per grid.

Finally, this subsurface model does not completely replace diffuse. Ideally, the falloff function would spike at a distance of zero, more closely modeling the properties of real skin. However, this is computationally inefficient since many more samples would be required to accurately model high frequency diffuse detail. Instead, our falloff function flattens out near the point being shaded and the subsurface illumination is mixed with diffuse.

5.3.3 Implementation

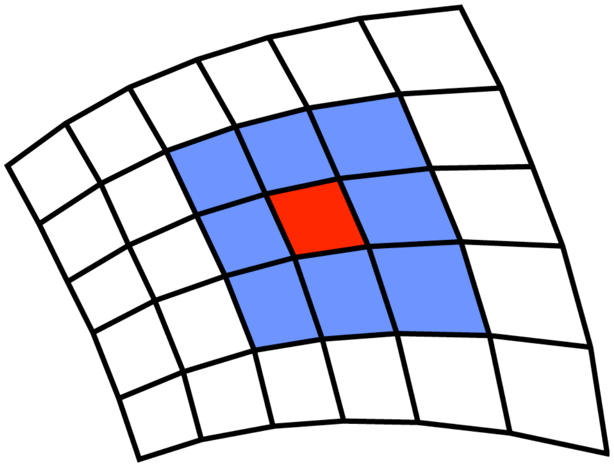
Below is a shading language implementation of this grid counting subsurface illumination approach.

```
color Scatter(color  Cbase;
              point  P;        /* The point being shaded */
              point  Porig;    /* P before displacement  */
              normal N;
              float  Kd;
              string mapname;
              float  scatterWidth;
              color  scatterTint;
              output float __overrideDerivs;/* Message passing to the light shader
*/
              output float __derivA;)       /* Use this area for the filterwidth*/
{
    color Idiffuse = Diffuse(P,N);
    color Iscatter = 0;

    varying point  Pworld  = transform("world",Porig);
    varying normal Nworld  = normalize(ntransform("world",N));

    /*
     * Read in any maps that haven't been read
     * Keep pointers to them for future calls
     * Free any cached information from the previous grid.
     */

    ScatterDSOReadMap(mapname);

    /*
     * For each grid point, search the octree and create
     * a cache of nearby samples. Return the grid point's
     * index into this cache. At the same time, build a
     * queue of nearby samples that still need to be shaded.
     */

    float grid_index = ScatterDSOAddGridPoint(Pworld,
                                              scatterWidth);

    /* Irradiate all samples in the queue, filling
     * the grid with more samples if necessary.
     *
     * prman will run the light shader on all grid points
     * in this grid, even if called on only one grid point.
     * If we were to serialize this portion of code (by
     * running only on the first grid point, it would thus
     * be extremely inefficient.) To thwart this, we'll always
     * fill an entire grid. If there aren't enough samples
     * in the queue, we'll use any old samples we can find.
     * (cause we'll probably need to shade them eventually)
     */

    varying point  Psample = point(0,0,0);
    varying normal Nsample = normal(0,0,0);
    varying float  Asample = 0;
    varying color  Isample = color(0,0,0);

    __overrideDerivs=1;

    while(ScatterDSOGetSample(Psample,Nsample,Asample)) {
        __derivA = Asample*length(transform(“world”,“current”,
                                          vector(1,1,1));

        Isample = Diffuse(Psample,Nsample);

        ScatterDSOSaveIrradiance(Isample);
    }

    __overrideDerivs=0;

    /*
     * Accumulate illumination from the nearby
     * samples, using the cache to avoid searching
     * the octree again.
     */

    Iscatter = ScatterDSOScatter(grid_index,Pworld,
                                 Nworld,scatterWidth,scatterTint);

    return Kd*Csurf*mix(Idiffuse,Iscatter,scatterAmount);
}
```

5.4 Point Cloud Generation

There are several techniques for generating a point cloud suitable for scattering using the methods described in the previous sections. The one I developed for “Finding Nemo” leverages RenderMan to do much of the work. The approach is somewhat similar to a texture atlas and is designed to work specifically with subdivision surfaces.

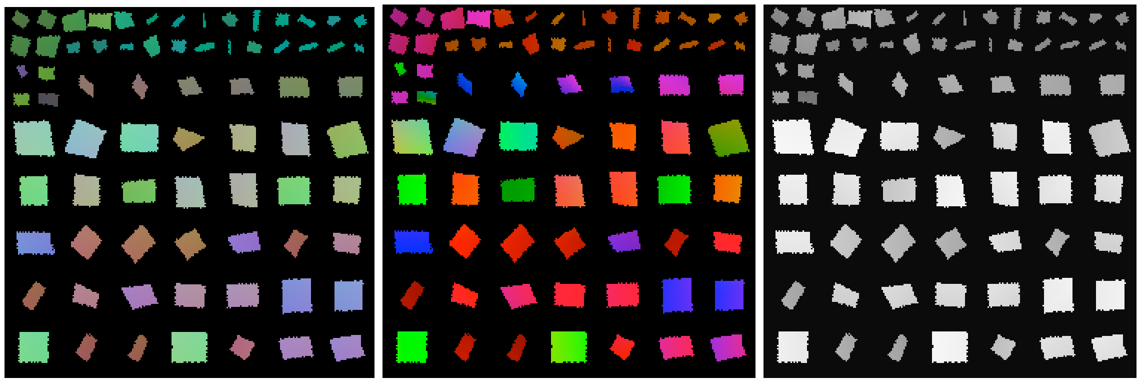
All the faces in the mesh are laid out in the camera plane and rendered to produce maps of positions, normals, and areas per sample. These maps can then be read directly by the DSO to produce the octree. Consider a mesh such as this.



When rendering the “scatter map”, each face needs to be oriented into the camera plane. To do this a normal for the face is found and used to construct a linear transformations that rotates the normal to align with the camera. Next the bounding box of the largest face in camera space is computed and used to uniformly scale the faces so that they can be laid out in a unit grid.

In addition to the face itself, the new meshes must also contain the adjacent faces, marked in blue above. This is because the subdivision algorithm needs a two vertex area of support in order to produce accurate points and normals. The area represented by the blue faces will not actually be rendered.

So, for a mesh with n non-border faces, n new meshes are created. The mesh is passed through a RenderMan procedural “RunProgram” to convert the single mesh into the n new meshes. After rendering, which typically takes only a few seconds, three images such as those below are produce. The first represents the point data, the second represents the normals, and the third represents the micropolygon area of each sample. All three are represented in “world” space.



Since these maps are not really images so much as collections of point samples, it’s important to set a few options and attributes in the RIB to disable filtering and quantization. RenderMan’s arbitrary output variables are used to write all three images at once. The images are written in 32-bit floating point format with LZW compression.

```
PixelSamples 1 1
PixelFilter “box” 1 1
Quantize “rgba” 0 0 1 0
Display “Model.tiff” “rgba” “compression” [“LZW”]
Display “+ModelP.tiff” “varying point ScatterP” “compression [“LZW”]
Display “+ModelN.tiff” “varying normal ScatterN” “compression [“LZW”]
Display “+ModelA.tiff” “varying float ScatterA” “compression [“LZW”]
```

To lay out the meshes on the camera plane regardless of the camera’s position or orientation, we set the current coordinate system to “NDC” just before calling the procedural to generate the meshes. The procedural can then simply place the meshes on the unit square. Here are the RIB commands that define the mesh.

```
CoordinateSystem “actualobject”
Surface "Humans/Skin/Bake" "uniform string ScatterActualObject" ["actualobject"]
TransformBegin
CoordSysTransform "NDC"
Procedural "RunProgram" ["fragment" "mesh data…”] [0 1 0 1 0 1]
TransformEnd
```

Of course, we wish to generate maps that encodes the original mesh, not for the flattened mesh being rendered. To do this, the procedural program, fragment, attaches a reference mesh with the original points being rendered.

So far so good, but there is a problem. RenderMan considers the mesh’s “object” space to be the current transformation when the mesh is defined. In this case, that’s “NDC”. But the points of our original mesh are defined the space just before we switch to “NDC.” If we want to write out our maps in “world” space (which allows the maps to be camera independent) we need to somehow get this real object space to the shader.

To do this, we define a new coordinate system, “ModelActualObject”, before replacing it with “NDC.” The points of the mesh will be transformed by RenderMan from “object” to “current” space, so to get to “world” space we must:

- Undo the “object” to “current” transformation.
  Points are now as they were declared.
- Transform into “world” from the real object space, “actualobject”.

RenderMan will complain about this, since it believes that we’re transforming “object” space points from “actualobject” space. ‘#pragma nolint’ will suppress the compiler warning. This all works well as long as an orthographic camera is used.[6] Here’s the shader used to generate the scattermaps.

```
surface Bake (
           varying point  ScatterPref         = point(0,0,0);
           uniform string ScatterActualObject = "actualobject";
           uniform float  ScatterVis          = 1;
   output  varying point  ScatterP            = point(0,0,0);
   output  varying normal ScatterN            = normal(0,0,0);
   output  varying float  ScatterArea         = 0;
) {
   /*
    * This transforms ScatterPref into “current” space as
    * if it were specified in "actualobject" space
    * rather than “object” space.
    */

   point Psurf = transform("object",ScatterPref);
#pragma nolint
   Psurf = transform(ScatterActualObject,"world",Psurf);
   normal Nsurf = normalize(calculatenormal(Psurf));

   /*
    * Save the point and normal in worldspace.
    * Save the worldspace area of this sample.
    *
    */

   Scatter02P    = transform("current","world",Psurf);
   Scatter02N    = normalize(ntransform("current","world",Nsurf));
   Scatter02Area = area(Scatter02aP);

   Ci = Oi = ScatterVis;
}
```

5.5 Possible Extensions

There are several possible extensions to this system. The first is the use of a ‘face adjacency’ mapping in place of the octree for quickly locating nearby samples. Since the scatter map baking system described in section 6 operates on faces, it would be easy to generate a fourth map that encodes a unique id for each face. The DSO would then generate a world-space bounding box and adjacency mapping for each face. The adjacency mapping for a face F would simply be a list of all other faces in the map that have a bounding box that is a distance of D or less away from the bounding box of F.

There are a couple of potential advantages of this approach. First, the amount of memory required to represent an octree data structure is fairly high and grows with the number of

[6] The projection in a perspective camera will cause the “object” to “current” transformation to not be invertible.

samples. A face adjacency map, on the other hand, requires only a fixed amount of memory for the data structure (based on the mesh size and it's topology) plus the space for the actual sample data. Therefore a face adjacency map uses memory more efficiently than an octree when the scatter maps are rendered at higher resolutions.

Maintaining the division of samples into faces also provides a natural structure on which to implement a least recently used (LRU) cache system. Loaded into memory, a scatter map's octree can easily take up several hundred megabytes of memory. Along with other memory intensive elements of the scene, particularly hair, our renders on "Finding Nemo" would frequently fail due to RenderMan processes reaching the 2Gb memory limit of 32bit machines. To combat this, an ordered list of faces could be kept. All but a certain number of most recently accessed faces would be written out to disk, then read back in as necessary.

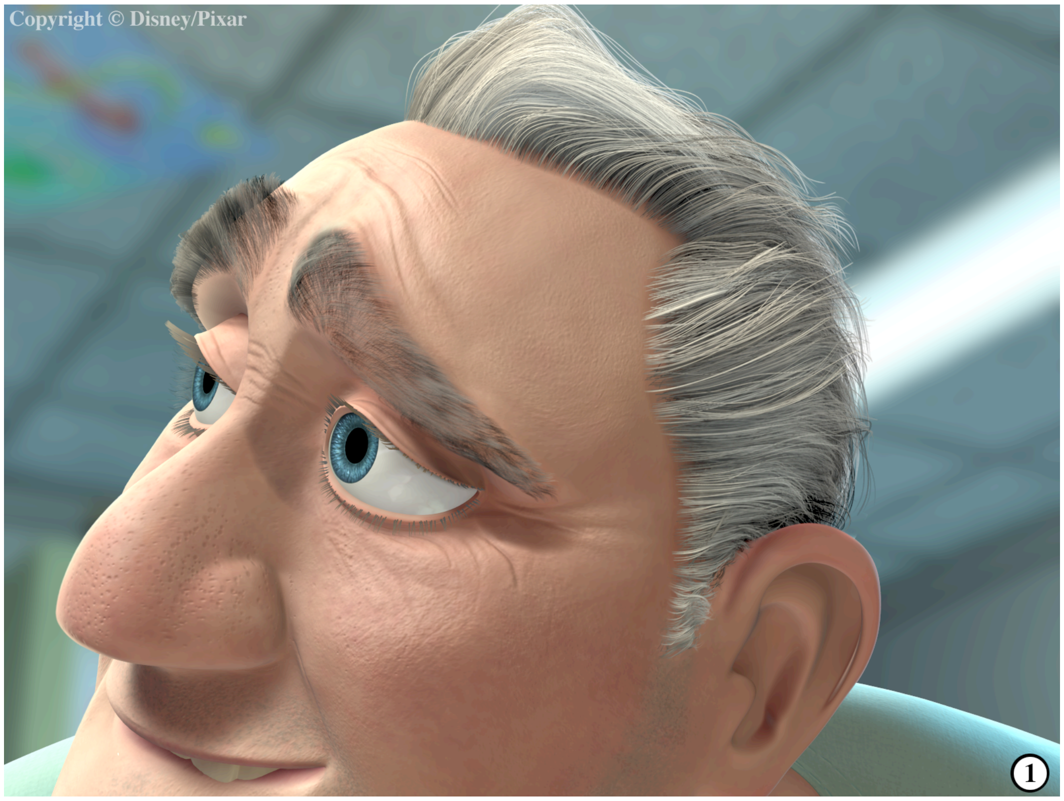
Finally, parts of this system have applications outside of subsurface scattering. In a general sense, this system gives the shader access to points that are near the point being shaded. This is a fundamental ability that can potentially enable a wide range or techniques and applications. For instance, the Voronoi noise function frequently used in shading generally requires either that the surface have a good parameterization or that the noise be computed in three dimensions. However, a simple extension of this system could make the implementation of Voronoi noise across an unparameterized manifold quite trivial.

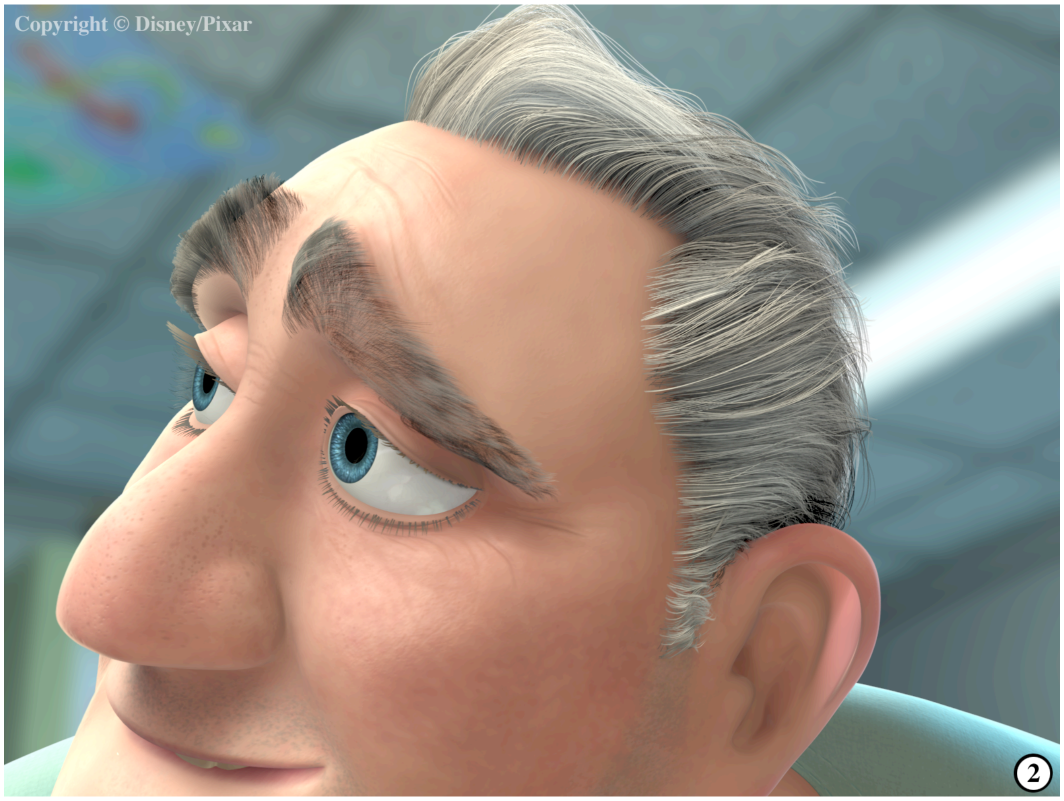
5.7 Acknowledgements

I'd like to thank my supervisors, managers, and producers on "Finding Nemo" for supporting this project and taking the risk to go ahead with it so late in production. Daniel McCoy helped me immeasurably with a variety of technical RenderMan-specific hurdles. Henrik Wann Jensen's 2001 and 2002 Siggraph papers on the topic were the inspiration that got me started.

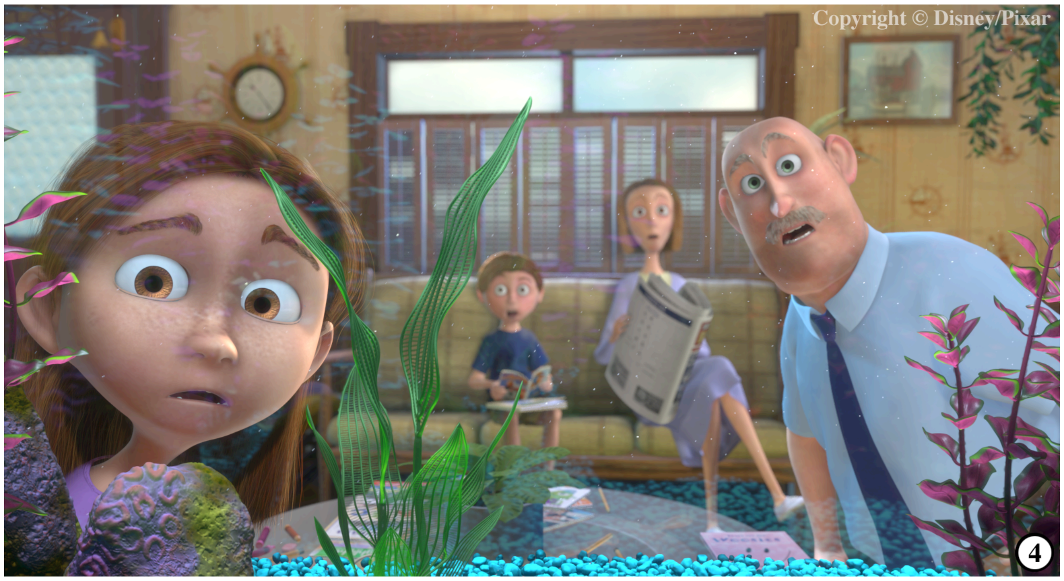
5.6 Results

Image 1:	The dentist rendered with BRDF illumination only.
Image 2:	The dentist rendered with subsurface scattering.
Images 3-5:	Subsurface scattering on other humans characters from "Finding Nemo."


Copyright © Disney/Pixar
1


Copyright © Disney/Pixar
2

3



4

5

Chapter 6

Translucency Tricks

Wayne Wooten, Pixar Animation Studios
wlw@pixar.com

6.1 Introduction

Materials capable of transmitting light are normally considered transparent; one can see through them. In contrast, materials that absorb or reflect all incident light are considered opaque. All bulk metals fall into the opaque category. (Although research has led to the development of porous metals that can transmit light and may one day eventually produce the fabled transparent aluminum oft alluded to in "Star Trek IV: The Voyage Home"). Glasses, crystals, polymers, and other inherently transparent dielectric materials fall into the transparent category. Translucent materials fall somewhere in between these two and transmit light diffusely. Light can either be scattered and absorbed at the surface or in the interior of translucent materials. The end result is that objects behind a translucent material cannot be clearly viewed through the material.

Most non-metallic materials will exhibit some amount of subsurface scattering. Recent algorithms in the computer graphics literature have given us models of BSSRDF and have even provided efficient methods for modeling this particular phenomena. BSSRDF provides a more photorealistic reproduction of materials that have a soft appearance such as marble, milk, and skin[3, 5, 4]. However many of these materials would be considered opaque given the definition above because other objects cannot be resolved when placed behind these types of soft objects.

This chapter will concentrate on modeling the effects of translucent materials where one can see another object behind the translucent object, for example sheets of tracing paper, sheer curtains, clear ice, frosted glass, or thin leaves. Thicker objects that do not have a very high extinction coefficient (little to no indirect multi-scattering) can also be modeled with the techniques presented here. Figure 6.1 shows some photographs of real world objects that provide inspiration when attempting to reproduce this type of translucent effect.

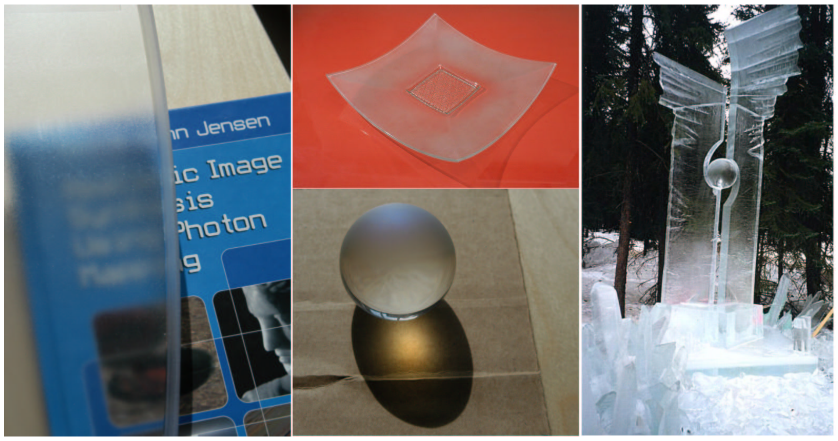


Figure 6.1: Translucent objects from the real world. Clockwise from the right, an ice sculpture, a diffuse caustic from a half frosted sphere, a rough plastic surface, a frosted glass bowl.

The next section provides a model of surface scattering translucency and provides an implementation using the ray tracing features of the shading language. The third section will describe how a new feature of PhotoRealistic RenderMan can be used to achieve translucency effects in an efficient timeframe without tracing rays.

6.2 Translucency Using Ray Tracing

One way to model the effect of translucency in RenderMan is to use the ray tracing features of the shading language. The surface at the point we are trying to shade is rough, meaning transmission rays through the material are mostly incoherent. This incoherency can also be caused by imperfections inside the material, but we will generalize to imperfections on the surface for our examples.

Because the rays are incoherent at the shade point P, we would like to integrate the contribution of all the different ray hits on the surfaces behind the translucent surface to derive the final color at the point P[2]. This average of the hit point colors gives the final blurry result. Listing 6.2.1 is an example of a shader that produces a translucent effect using the `gather()` call for ray tracing.

Listing 6.2.1 Example shader using `gather()`

```
surface translucent ( float period = 4, noisy = 1, var = 1, amp = .5;
                      float Ks = .5, Kd = .5, Ka = 1, roughness = 0.1;
                      color specularcolor = 1;
                      float angle = 2.5, samples = 128)
{
    vector Vn = -normalize(I);
    normal Nf = faceforward(normalize(N), I);
```

```
color raycolor = 0;
color sum = 0;
float dist = 0;

float sqrt3 = sqrt(3);
point Pshade;
float blob;
color Cdiff, Ctrans, Cspec;

float hexgrid(float u, v)
{
    float uu, vv;
    extern float sqrt3;
    float l(float x, y){ return sqrt(x*x+y*y); }

    uu = mod(u, 1);
    vv = mod(v, sqrt3);
    if (vv>.5*sqrt3) vv = sqrt3-vv;
    return min(l(uu, vv), l(uu-1, vv), l(uu-.5, vv-.5*sqrt3));
}

float snoise(float u, v)
{
    return float noise(u, v)*2-1;
}

float noisygrid(float u, v, period, noisy, var, amp)
{
    return hexgrid(period*u+amp*snoise(noisy*period*u,noisy*var*period*v),
                   period*v+amp*snoise(noisy*var*period*u,noisy*period*v));
}

/* Compute blob areas where object is opaque or translucent */
Pshade = transform("shader", P);
blob = smoothstep(.52, .54, 1-noisygrid(xcomp(Pshade), ycomp(Pshade),
                  period, noisy, var, amp));

/* Compute direct illumination (ambient, diffuse, specular) */
Cdiff = Cs*(Ka*ambient()+Kd*diffuse(Nf));
Cspec = specularcolor*Ks*specular(Nf, Vn, roughness);

/* Compute translucent component */
gather("illuminance", P, -Vn, radians(angle), samples, "surface:Ci", raycolor)
{
    sum += raycolor;
}
Ctrans = sum / samples;

Oi = Os;
```

```
      Ci = Oi*(mix(Cdiff, Ctrans, blob)+Cspec);
  }
```

This shader is composed of three parts. First it computes a blob value that will be used to select areas of the surface that are either opaque or transparent. Then it computes the direct illumination of the surface which will be used for the opaque part of the surface. Finally it computes the translucent part of the surface and uses `mix()` to select which part of the surface will be opaque or translucent. Figure 6.2A is an example that shows the use of this shader. The efficiency of the shader could be improved by only computing the translucent component in areas where the blob value indicates the surface should be translucent. I leave this as an exercise for the reader.

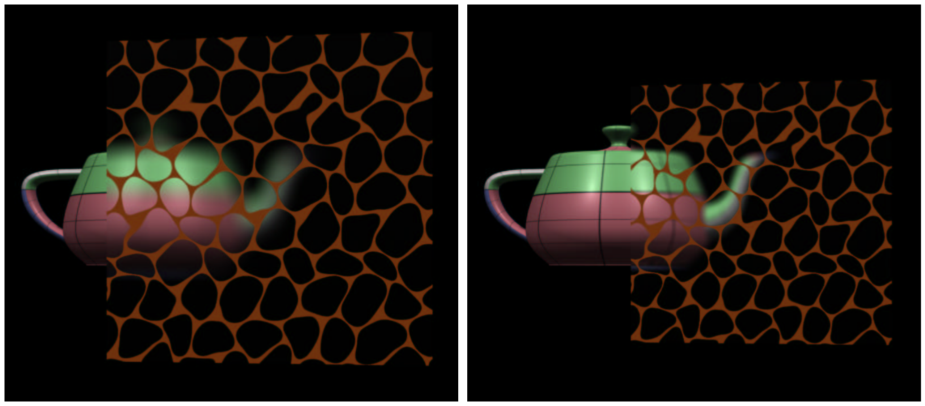


Figure 6.2: **A.** The left teapot sits behind a translucent membrane. **B.** The teapot on the right is less blurry, as the translucent membrane is closer to the object.

The key function in this simple example is `gather()`. This function is a looping construct that fires a number of rays (defined by `samples` in this case) from point `P` on the surface. The direction of the rays is defined to be the negative of the incident ray from the eye to the surface, `-Vn`. This shader assumes that light is not refracted as it travels through the surface and that the translucent effect is view dependent. Each of the nsample rays are not fired in the exact same direction though, because the cone angle parameter (`radians(angle)`) is non zero. The nice thing about `gather()` is that it will jitter the ray directions for each sample ray within the solid-angle derived from the half-angle value given by the cone angle parameter, `radians(angle)`. As the cone angle is increased, the translucent surface will exhibit more blur.

6.2.1 Pros and Cons

There are three noteworthy advantages of using ray tracing to create this effect. The main benefit is that the shader automatically takes into account the effect that as an object behind a translucent object moves closer to the translucent object, the background object exhibits less blur. This happens because the dispersal area of the rays leaving the translucent object decreases as the distance between the translucent object and the background object decreases. An example of this can be seen in figure 6.2B.

Another advantage of this approach is that multiply layered translucent objects will correctly accumulate blur. As rays from one translucent object hit another translucent object, the second object will then spawn rays to calculate its own color, before returning that color to the first translucent object. An example of this can be see in figure 6.3. A neat trick that can be used to speed up multiple layers is to use the built in function called `rayinfo()` to obtain the ray depth at the hit point. If the depth is non zero, that means the point in question is being hit by a secondary ray and can shoot fewer samples than a point hit by a primary or camera ray.

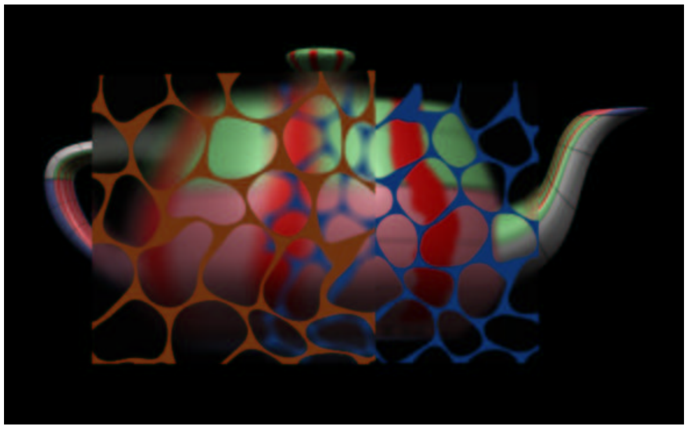


Figure 6.3: A teapot behind two different translucent planes.

The third advantage of this method is that translucent objects correctly influence the lighting on background objects by blurring the change in lighting as well. An example of this is shown in figure 6.4 where the shadow cast by the red cylinder on the ground plane is in turn blurred by the translucent object between the cylinder and the ground plane.

In summary, the advantages of using ray tracing to achieve this affect are that it is trivial to add a `gather()` call to a shader, it does not require multiple rendering passes, and it captures most effects one would expect from a translucent material. However, the single disadvantage is the render time. This is especially true if there are multiple layers of translucent objects, or the background behind the translucent object is very complex. For the simple teapot scenes in this tutorial, the run time was significantly slower than the technique that will be presented in the next section.

6.3 The Fine Art of Chi'Ting

In a SIGGRAPH course notes presentation the 80's, Jim Blinn made a statement that still very much rings true today[1]. Basically he said that whenever we try to simulate a complex physical phenomena, we will soon realize that an accurate simulation is far beyond the capabilities of today's modern computing hardware. Today's hardware is exponentially more powerful than that available

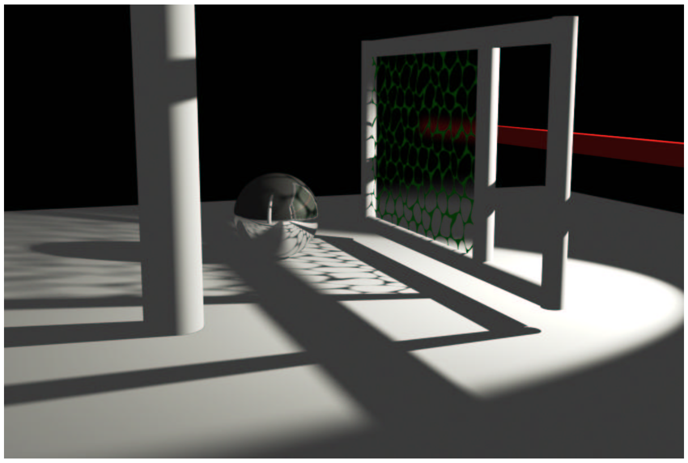


Figure 6.4: A translucent membrane blurs the shadow of the red cylinder.

in the late 80's, but the truism still holds that accurate simulations of the real world are not really necessary to make believable pictures.

In this section we will describe how a translucent effect can be created without resorting to simulating the rays of light that are scattered by a translucent surface. The algorithm uses a new feature of PhotoRealistic RenderMan to create a depth mask, recording the depth of the closest hit of each translucent surface at a subpixel resolution. This mask can then be used to render a background image that can then be applied as a blurred texturemap in the translucent object's surface shader. This section shows how this can be done with a single layer of translucent objects in the scene. Repeating the algorithm for multiple layers yields acceptable results as well.

6.3.1 Using Depth Masks

A new hider option has been added to PhotoRealistic RenderMan that hides objects that are either in front of or behind a given depth value from a shadowmap. This option can be specified as:

```
Hider "depthmask" "zfile" "MAPNAME.smap" "reversesign" [0|1] "depthbias" [float]
```

This tells RenderMan that we want to use the specified shadowmap to alter how objects are hidden from the camera. If the `reversesign` option is `0` (the default) then the hider will remove any geometry that is in front of the depth map. If `reversesign` is `1`, then any geometry behind the depth map is culled.

The one restriction for using the `depthmask` hider is that the mask must be rendered at sample resolution. This helps eliminate aliasing issues that might arise when comparing the depth of the sampled geometry to the value in the depth map. Of course this means the depth map will have a resolution `PixelSamplesX x FormatX` by `PixelSamaplesY x FormatY`. One can cause RenderMan to create sample resolution images using the `subpixel` option of the normal hider.

```
Hider "hidden" "subpixel" [1]
```

One final option that can be used with the `depthmask` hider is `depthbias`. This parameter controls the amount of bias applied to the mask before checking against the rendered geometry. The default for this parameter is 0.01. Raising this value will prevent self-intersection in the case where the mask and the rendering geometry coincide. In other cases where two surfaces are extremely close, a lower `depthbias` value will prevent the rendering geometry from being culled.

6.3.2 Depthmasks for Translucency

The first render pass involves creating a shadowmap that can be used with the `depthmask` hider on the subsequent passes. The `mask` phase of the shader in listing 6.3.1 is used in this pass to simply set the opacity of the translucent areas of translucent foreground object. No background objects are rendered in this pass. An example shadowhmap can be seen in figure 6.5A.

Listing 6.3.1 Example shader using `depthmask`

```
surface translucent ( float period=4, noisy=1, var=1, amp=.5;
                      float Ks = .5, Kd = .5, Ka = 1, roughness = 0.1;
                      color specularcolor = 1;
                      string background = "must be set in RIB file";
                      float rad = .01;
                      string phase="mask")
{
    /* This shader uses functions defined in Listing 6.2.1 */
    Pshade = transform("shader", P);
    blob = smoothstep(.52, .54, 1-noisygrid(xcomp(Pshade), ycomp(Pshade),
                      period, noisy, var, amp));

    /* In the "mask" pass simply mark the translucent coverage */
    if (phase=="mask"){
        Oi = blob;
    }
    /* In the "behind" pass shade the translucent object */
    else if (phase=="behind"){
        Nf = faceforward(normalize(N), I);
        V = -normalize(I);
        Oi = 1 - blob;
        Ci = Oi*(Cs*(Ka*ambient()+Kd*diffuse(Nf))+
                 specularcolor*Ks*specular(Nf, V, roughness));
    }
    /* In the "final" pass shade with a blurry NDC texturemap lookup */
    else if (phase=="beauty"){
        point Pndc;
        float x, y;
        color Cdiff, Ctrans, Cspec;

        Nf = faceforward(normalize(N), I);
        V = -normalize(I);
        Cdiff = Cs*(Ka*ambient()+Kd*diffuse(Nf));
        Cspec = specularcolor*Ks*specular(Nf, V, roughness);
        Pndc = transform("NDC", P);
        x = xcomp(Pndc);
        y = ycomp(Pndc);
        Ctrans = texture(background, x-rad, y-rad, x-rad, y+rad,
                         x+rad, y-rad, x+rad, y+rad);
        Oi = Os;
        Ci = Oi*(mix(Cdiff, Ctrans, blob)+Cspec);
    }
}
```

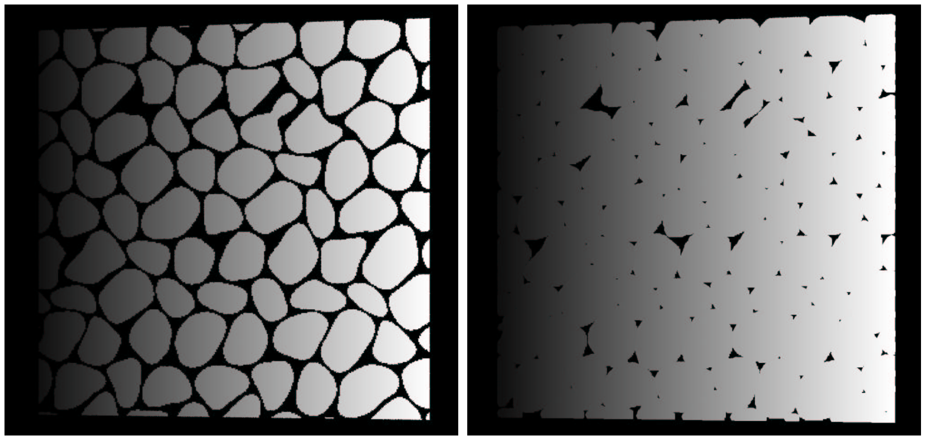


Figure 6.5: **A.** The depth mask on the left. **B.** The depth mask after dilation has been performed.

If this depth image was simply used as rendered, the result would not be acceptable. This is because the background objects will be blurred to simulate the translucent effect and this blur requires an area of support. If we do not consider rendered pixels in the background image from a region extending beyond the boundaries of the translucent object, the resulting blur will be dark around the edges. We solve this problem by dilating the boundaries of the depth mask so that we include areas in the background that should contribute to the blur. Listing 6.3.2 gives the pseudo-code for a dilation algorithm. An example of the depth mask after dilation can be seen in figure 6.5B.

Listing 6.3.2 Pseudo code for depth mask dilation

```
Read Image
blurRadius = desiredRadius/NumPasses * imageSizeX
ForEach (Pass)
   Mark Boundary Pixels
```

```
    ForEach (Boundary Pixel)
       ForEach (Neighbor Pixel in +/- blurRadius)
           Propogate depth of Boundary Pixels to Neighbor Pixels
 Save Image
```

6.3.3 Background Pass

The next step in the process is to render a background image that will serve as a NDC (normalized-device coordinates) texturemap in the final pass. The `depthmask` hider ensures that only the objects behind the translucent object are used to create the texturemap. Only rendering the objects behind the translucent object serves a couple of purposes. First, objects in front of the translucent object should not contribute to the texturemap or the effect would be incorrect. Second, objects that are not behind the translucent object will be occlusion culled early. Micropolygon grids from the culled objects will not be shaded and the render time will be faster.

If the translucent object does not hide itself, it does not need to be rendered in this pass. It would be culled anyway and by removing it, potential bias problems are avoided. If the topology of the object prevents it from being discarded, then the `behind` pass of the shader in listing 6.3.1 is used to shade the translucent object as if it were simply a transparent object. Figure 6.6A shows the background image that results from using the dilated depth mask. Figure 6.6B shows a background image that results from using a more complex translucent object (another teapot in this case).

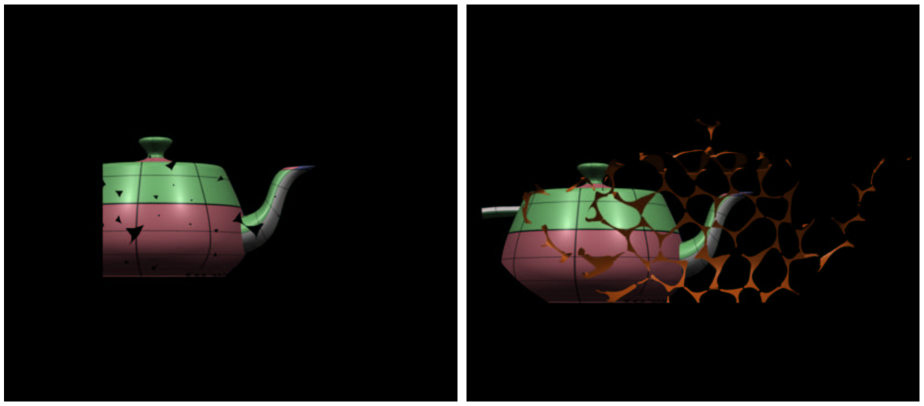


Figure 6.6: **A.** The background pass with mask applied on the left. **B.** A complex background on the right includes the translucent object itself.

6.3.4 Beauty Pass

The final step involves taking the image from the background pass and converting it into a texture map. The whole scene with both translucent object and all objects behind the translucent object are rendered. This time the shader in listing 6.3.1 uses the `beauty` pass to render the translucent object. Note that the object's opacity in this pass is no longer affected by the shader. Instead in the areas that are considered translucent, a four point texture call is made that results in a blurry texture lookup. Since this texture lookup is made in NDC space, the blurred background objects correspond with the sections of the background object that are not behind the translucent object. The larger the `rad` parameter to the shader, the more blurry the translucent object appears. Figure 6.7A shows the final result using the image from the background pass as the NDC texturemap. Figure 6.7B shows the final pass for the more complex case.

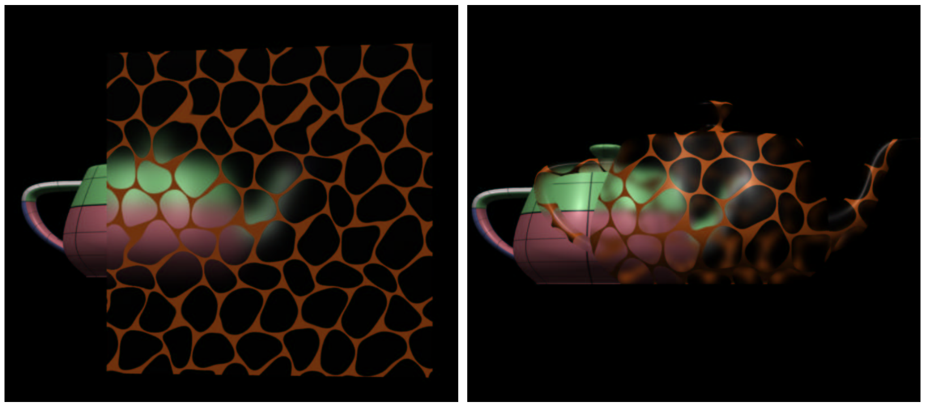


Figure 6.7: **A.** The final result on the left. **B.** A more complex final result on the right.

6.3.5 Pros and Cons

The main advantage with the multi pass technique is speed. RenderMan is fast at rendering depth maps, mip-mapping the background image ensures quick blurs on texture lookup, and rendering the final pass is also relatively fast. The speed of this method was a key factor in deciding to use this method to render the jellyfish elements in Pixar's animated feature, *Finding Nemo*.

There are a couple of disadvantages to this method. The primary disadvantage is the complexity of the pipeline. Multiple passes have to be made over the RIB files and various pieces of geometry have to be removed during some of these passes. Animated sequences have to keep track of the depth masks and texturemaps that are created for each frame. And the pipeline becomes increasingly more complex if one needs to perform multiple layers of translucency.

Another disadvantage of this method is that one has to mimic the effect that as an object is closer to the translucent object, it exhibits less blur. This involves computing the distance between the translucent object and the objects behind it. By storing another shadowmap of the background objects with the texturemap, the shader in listing 6.3.1 could be easily modified to take this into account. It is much more difficult to obtain the effect where the translucent object modifies the sharpness of shadows on objects behind the translucent object.

6.4 Conclusion

This tutorial presents two different methods for creating the look of translucent objects that a viewer can see through. The first method uses ray tracing with the `gather()` call and easily generates most of the expected effects. Ray tracing is slow however, so a second method is presented that runs much faster using a new `depthmask` hider. This technique requires a more complex pipeline and shaders, but the performance benefits are worth the effort if an accurate translucent effect is not required.

6.4.1 Acknowledgements

Thanks to all the wonderful TDs and developers at Pixar who put in more work than I to make depth mask translucency possible. In particular, I would like to thank Tom Duff who originally developed the multi-pass approach. I would also like to thank Susan Fisher who tirelessly implemented pipeline tools and improved RenderMan to make the multi-pass approach viable in a production setting. Thanks to Dave Batte who managed to use all of our code in a feature length film. And finally thanks to the RenderMan Products Group who developed ray tracing in PRMan and made rendering fun again.

References

[1] James F. Blinn. Technical note: The ancient chinese art of chi-ting. *Jet Propulsion Laboratory*, January 1988.

[2] Robert Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Computer Graphics (Proc. SIGGRAPH 84)*, pages 137–145. ACM, 1984.

[3] Henrik Wann Jensen. *Realistic Image Synthesis using Photon Mapping.* A K Peters, 2001.

[4] Henrik Wann Jensen and Juan Buhler. A rapid hierarchical rendering technique for translucent materials. *ACM Transactions on Graphics*, 21(3):576–581, July 2002.

[5] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 511–518, August 2001.

Chapter 7

Shading Fish Guts

Byron Bashforth
Pixar Animation Studios

Introduction

In *Finding Nemo*, an odd-ball assortment of fish are residents of a dentist's aquarium. One of these fish, Bubbles, is a yellow tang. Yellow tangs, in the real world, are bright, tropical fish that are surprisingly thin and translucent. When lit from behind, you can easily see the skeleton running the length of the fish and a large, silvery-green bag of guts behind the head.

Replicating the appearance of the real fish would be too distracting for this film (especially the bag of guts right behind the eyes - imagine talking to someone and being able to see their liver floating in their head) but we needed to convey Bubble's essence of being thin.

This paper discusses the technique we used to shade the internal structure of Bubbles and portray him as a translucent fish. Section 1 discusses the key features that need to be addressed while Section 2 describes our solution. We conclude with our final remarks in Section 3.

1. Features

To convince an audience that we are seeing the guts inside Bubbles, we need to address three key issues.

1.1 Parallax

If there are guts inside a fish, they will parallax correctly. (Parallax is the apparent shift of an object's location resulting from a change in the viewer's position.) Some early experiments on *Finding Nemo* used illumination tricks to make a fish feel like it had an internal structure wrapped in a fleshy gel. The innards were painted on the outside of the fish and then illuminated as if they were beneath the surface. The gel filtered the color of the innards over a simulated distance (and the gel's own illumination was added at the surface).

When looking directly at a flat portion of the fish, this is successful. As we move towards the edge, we expect the gut paintings to change more slowly than the surface detail. However, the gut paintings are attached to the surface and do not parallax. The problem is compounded as the fish animates – the edges move and the illusion of guts is quickly dispelled. This trick is useful for background characters but not for fish that are scrutinized more closely.

1.2 Selective Visibility

We want to see into Bubbles (or think we can) but not through him. Perhaps the most obvious solution is to create gut geometry and then make his skin partially transparent from the outside and opaque from the inside. This would allow us to peer into the fish without seeing through.

This is problematic in areas where there is folding or complex geometry (like around the eyes and base of the pectoral fins). As well, the inside of Bubbles is unlike any real fish. His model is a hollow shell with occasional body parts, like the teeth and tongue, suspended in space [Figure 1].

We need a means to select what internal parts are visible from the outside and what external surfaces we can see through.

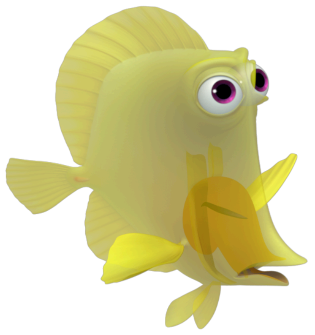


Figure 1: Bubbles with a 50% transparent skin. The internal geometry around the mouth and gills is obviously distracting. Also notice the folding around the eyes.

1.3 Diffusion

The flesh inside real fish diffuses light. Even if we manage to get guts inside the fish, a crisp render of them will make Bubbles look like a plastic toy with some sort of prize inside.

A volume shader is a possible solution, although a computationally expensive one. As we'll demonstrate in Section 2, our approach is much simpler.

2. Shading the Guts

Our solution involves projecting an image of gut geometry (an actual model) on to Bubbles with his skin shader. This addresses all of the issues of parallax, transparency, and diffusion.

There are three steps involved. The first requires a change to the model. The second and third are steps in a two-pass render.

2.1 Build a gut model

We require geometry inside the fish but it doesn't need to be detailed. For Bubbles, we constructed a low-res mesh that loosely approximates the skeleton, skull, and organs [Figure 2]. (Although not shown in Figure 2, we also treated the inside surface of the mouthbag as part of the guts. The mouthbag is the large shape attached to Bubbles' mouth in Figure 1.)

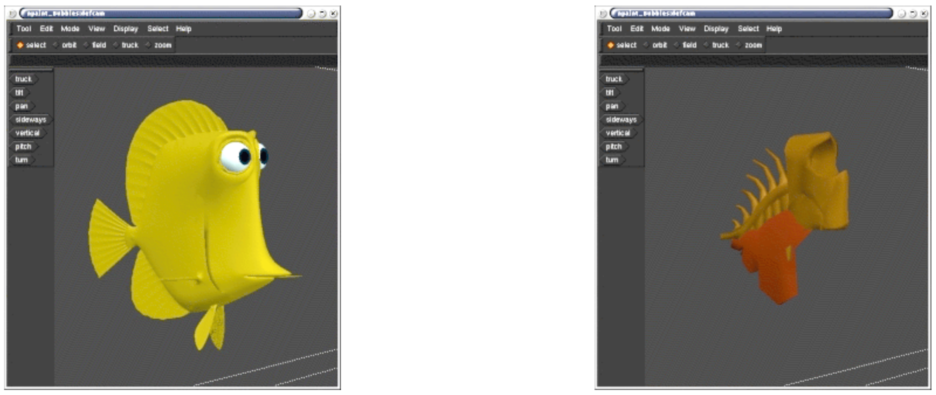

Tool Edit Mode View Display Select Help
select orbit field truck zoom
truck
tilt
pan
sideways
vertical
pitch
turn

Figure 2: Bubbles' skin and guts models.

2.2 Render the guts from the viewing camera and save the image in a texture map

This is the first step in the two-pass render. We turn off everything in the scene and render just the innards from the viewing camera. We discovered through exploration that the location of the guts is the only information we require. As a result, it is sufficient to render the image with a constant shader [Figure 3].

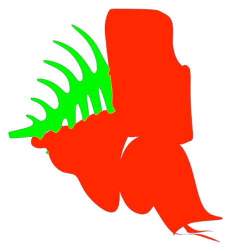


Figure 3: The gut map.

We did assign different colors to the guts - the spine is green while the other innards are red. The color choice is arbitrary but using red, green, or blue simplifies reading the information into the shader in Section 2.3.

It is important to reiterate that this texture map (the gut map) is rendered from the viewing camera and not from any camera that has a fixed relation to the character. As a result, the gut map must be rendered on every frame of the shot.

2.3 Render the fish and project the guts back on to the skin

Projecting the gut map on to the fish is simple. NDC space is the texture space of the viewing camera.

```
point texP = transform("NDC", P);
float ss   = xcomp(texP);
float tt   = ycomp(texP);

color Cguts = texture(gut_map, ss, tt, ...);
```

As we mentioned earlier, a crisp projection of the guts looks unnatural. We need to blur the texture map lookup to fake diffusion. Of course, there are a few problems.

The blur is specified in texture space as a percentage of the image size and the texture is rendered from the viewing camera. A fixed blur is not correlated to the screen size of the fish. For example, a 10% blur when the fish is full screen has very different results than a 10% blur when the fish is a few pixels in the background. One can imagine a worst-case scenario where Bubbles swims from the background to the foreground while his guts become disproportionately sharper as he gets closer.

The following code snippet allows us to specify a blur in "fish space" and convert it to NDC space.

```
vector vProj = (1,1,0);
vProj = vtransform("current", "shader", vProj);

vProj = normalize(vProj);
vProj = vProj * BLUR_AMOUNT;
```

```
    vProj = vtransform("shader", "NDC", vProj);
    vProj = vProj * (1,1,0);
    blur  = length(vProj);
```

We take an arbitrary vector, *vProj*, in camera space that has some x and y component and transform it to shader space. This yields a vector in shader space that has a projection of (1,1,0) on the camera plane. (We cannot begin with a fixed vector in shader space since the orientation of the fish may hide some length of the vector by pointing along the viewing direction.)

Next, we normalize *vProj* and multiply it by our blur constant, *BLUR_AMOUNT*. *BLUR_AMOUNT* is the control available to the end-user.

We then transform *vProj* to NDC space, ignoring any value that might have crept into the z-component. The length of *vProj* in NDC space is the correct blur size to pass to the texture system (with the following exception).

Since Bubbles is very thin, the amount of screen space he occupies changes dramatically if we are looking at his side or directly at his face. An appropriate blur for his side is too large for the head-on view so we implement a simple heuristic to compensate:

```
    vector vSide = (0,1,1);
    vSide = vtransform("shader", "current", vSide);
    blur  = mix(blur * THIN_MULT, blur, abs(xcomp(vSide)
               * ycomp(vSide)));
```

We take a vector in shader space that has components in y and z (the long dimensions of the fish). If that vector, once transformed to current space, has a large component in x and y, we know we are looking at the flat side of the fish and should use the maximum blur. Otherwise, if the projection has zero length in x or y, we are looking at a thin edge of the fish and the blur should be reduced. (Bubbles has a value of 0.6 for *THIN_MULT*.)

There are other blur issues that we chose not to address. For example, 10% blurs in the x and y of NDC space have different results since a film frame is wider than it is high. Should Bubbles point his nose upwards and swim, the blur will change slightly.

We also discovered that faking diffusion requires very large blurs. This revealed an unexpected problem with texture mipmapping.

The texture format stores the original image at a resolution that is a power of two, like 1024 x 1024, and also stores copies, or mipmaps, of the image at resolutions of decreasing powers of two (512 x 512, 256 x 256, etc) [Figure 4]. A box filter is used to resize each of these mipmaps allowing smooth transitions between neighboring levels.

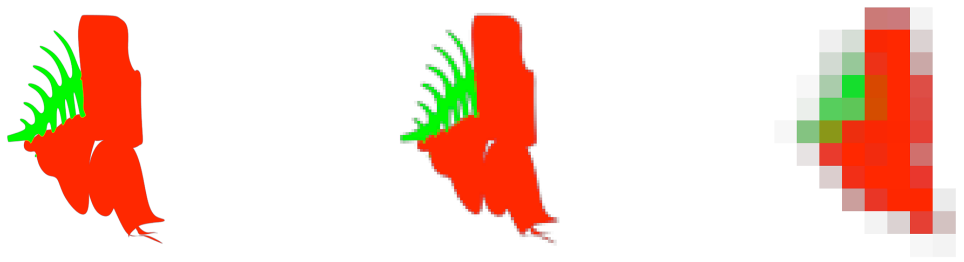


Figure 4: The gut map at 1024x1024 and two sample mipmaps resized to 256 x 256 and 32 x 32.

The smaller mipmaps are used when the texture filter width is large to save computation time. Typically, this happens when the object being textured is very small on the screen. There is little point processing an entire 1024 x 1024 texture map when the object is only 10 pixels big. In that case, using the 16 x 16 mipmap is probably sufficient.

For Bubbles, we are requesting very large blurs and generating large filter widths when the fish is full screen. Since the mipmap being used in the final render has been box-filtered, it has noticeable, blocky artifacts. As the fish animates, these artifacts are readily apparent.

Our solution is simple, although somewhat non-technical: We render the gut map at a very small resolution and disable mipmapping. We force the shader to filter the entire texture map but since the texture is small, the computation increase is negligible. Our experiments showed that the largest resolution we needed for Bubbles is 128 x 128 [Figure 5].

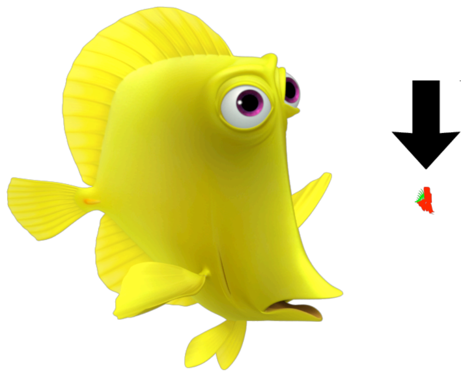


Figure 5: Bubbles and his 128 x 128 gut map.

Now we have a method to project the gut map back on to the fish and blur it sufficiently to fake diffusion. All that remains is using that information to change the look of the character.

First, we add a halo of color to fake diffusion around the guts. The code below demonstrates the idea:

```
float FgutsBlur = CALCULATEBLUR(GUTS_BLUR_AMOUNT);
float FskelBlur = CALCULATEBLUR(SKEL_BLUR_AMOUNT);

float Fguts = texture(gut_map[0], ss, tt, ..., FgutsBlur);
float Fskel = texture(gut_map[1], ss, tt, ..., FskelBlur);

Cskin = mix(Cskin, GUTS_COLOR, Fguts);
Cskin = mix(Cskin, SKEL_COLOR, Fskel);
```

We rendered the gut map with two colors [Figure 3] to distinguish between the skeleton and the rest of the guts. Now we can use that information to blur the skeleton less than the other innards (*GUTS_BLUR_AMOUNT > SKEL_BLUR_AMOUNT*).

Next, we darken the core of the guts to add a sense of thickness. Using code similar to the fragment above, we use smaller blurs on the gut projection and reduce the amount of backlighting that can pass through the skeleton and innards.

Finally, we disable this entire effect on the pectoral fins. Since they will cross in front of the body, the guts would be projected on to them as well (which is obviously the wrong thing to do). The completed fish is shown in Figure 6.

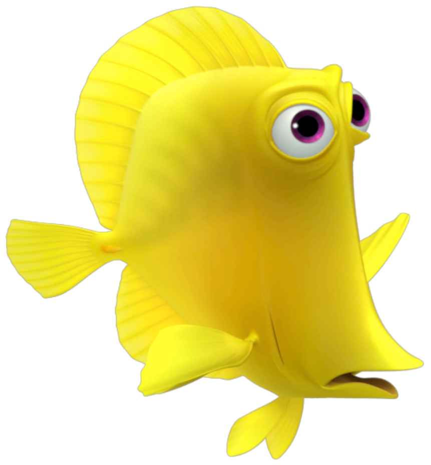


Figure 6: The final effect on Bubbles.

3. Conclusion

This technique provides an effective means to portray a very thin fish. Although it adds another step into the render pipeline, it only requires the generation of a small, constant-shaded texture map. The main challenge is the calculation of an appropriate blur since the gut map is rendered from the viewing camera (which gives us the parallax of the internal gut geometry). Once the blur is determined, projecting the guts on to the fish through the skin shader allows us to fake diffusion and control the visibility of the guts. For *Finding Nemo*, it enabled us to create one more unique character for the dentist's aquarium.

Thanks

Thanks to Daniel McCoy and Keith Olenick for their significant contributions to this work.

References

[Apodaca, Gritz 2000] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan.* San Francisco, CA: Morgan Kaufmann Publishers, 2000.

[Upstill 90] Steve Upstill. *The RenderMan Companion.* Reading, MA: Addison-Wesley, 1990.

Chapter 8

Reflections and Refractions in “Finding Nemo”

Thomas Jordan
Pixar Animation Studios

Introduction

In December of 2001, a group of three Technical Directors were assigned the task of developing reflections and refractions for the fish tank in “Finding Nemo.” Brad Andalman was assigned to the modeling aspects of the project, John Warren contributed his lighting expertise, and I represented our shading department. Over the next several months we met daily to brainstorm about the material properties of glass, how light reflects & refracts in nature, and how to simulate a believable fish tank using a computer.

Upon the completion of our reflections and refractions work, I was able to apply what we had learned toward the implementation of a second shader that simulated a bag of water containing a fish. I will now present our research and development of reflections and refractions for both the fish tank and the bag of water.

Research

Our first step was to acquire an actual fish tank. We filled it with pebbles, water, and a golf ball to act as Nemo’s stunt double. As we examined the tank we recorded our observations about how reflections and refractions varied depending on our viewing angle (from some angles we could see as many as 16 golf balls). We even submerged an underwater video camera to get an idea of what life looks like from the perspective of a fish. We were very surprised at how beautiful and complex something as simple as a fish tank can be.

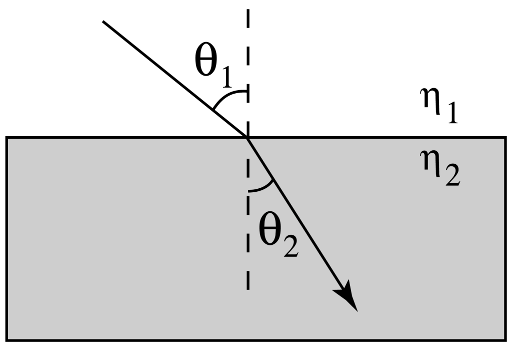
The more we learned about the fish tank, the more daunting our task became. The reflections and refractions we observed were very complicated and often confusing. This brought up an important question: How accurate do our reflections and refractions need to be? Most people have a *feeling* for how a fish tank should look, without knowing exactly how it’s *supposed* to look. And nobody knows how a fish perceives the world from the perspective of inside a tank. So we decided it was more important to make our CG tank believable as opposed to mathematically accurate. Not only did this make our task simpler, it gave us artistic freedom to bend the rules of physics.

Development

After several weeks of familiarizing ourselves with real life reflections and refractions, the time had come to test our theories and implement our observations. Our fish tank model was simple: four walls of glass, with each wall containing an inner and outer wall offset from each other to give the impression of thickness. The next step was to start shading our model.

Before shading began, we had to decide whether to use ray-tracing or multi-pass rendering. Ray-tracing may seem like the obvious choice because it calculates accurate reflections and refractions by definition. Unfortunately, the drawbacks of ray-tracing include slow render time (proportionate to quality), constrained results because of inherent mathematical accuracy, and (at the time) ray-tracing in *PRMan* was still in its infancy. Multi-pass rendering (using previously rendered images to gain knowledge about objects other than the current one being shaded) can be cumbersome to use and is less accurate than ray-tracing, but lends itself to artistic control. We made our decision and were able to achieve successful results in relatively little time using multi-pass rendering.

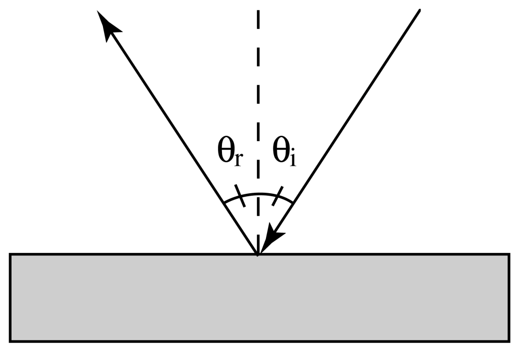
To implement reflections and refractions using multi-pass rendering, we needed to pre-render reflection & refraction images, which would later be used as texture inputs to our tank glass shader. We created these reflection and refraction images with reflection and refraction cameras. The default position of the refraction and reflection cameras was determined according to Snell’s Law and the Law of Reflection, respectively. The final positions of these cameras often changed, however, in order to create more artistic compositions.



Snell's Law

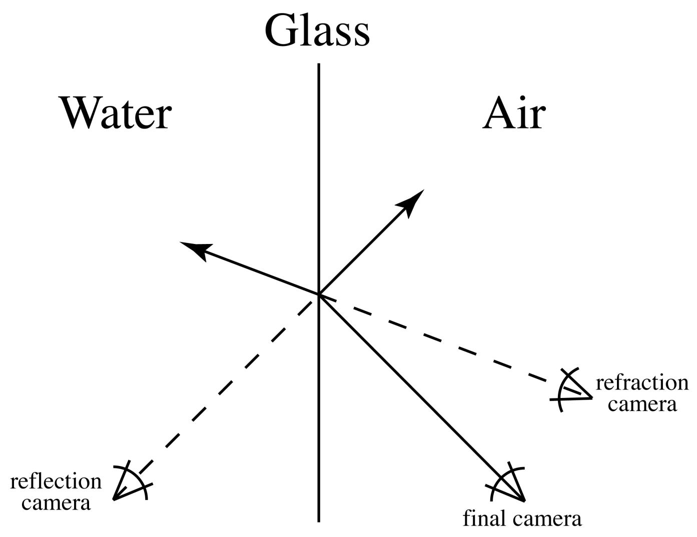
$$\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2$$

Snell’s Law describes the relationship between angles of incidence(□$_1$) and refraction(□$_2$) for a ray traveling between materials with different indices of refraction. In the context of the tank glass shader, □$_1$=1 for air, and □$_2$=1.33 for water.



Law of Reflection
$\theta_r = \theta_i$

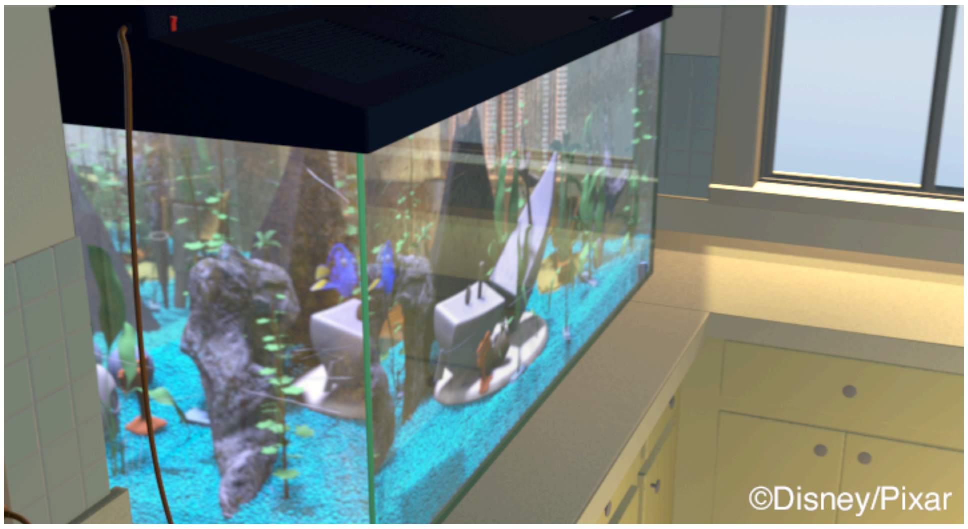
The Law of Reflection states that the angle of reflection(θ_r) equals the angle of incidence(θ_i). This means that the camera used to pre-render the reflection image is always on the opposite side of the glass from the camera used to render the final image. The following diagram shows the relationships between all three cameras and a single wall of the fish tank.



Each wall of the fish tank was assigned its own unique pair of reflection and refraction cameras, used for calculating reflection and refraction images specific to that wall. Our tank glass shader needed to be able to project these reflection and refraction images from the coordinate systems of the cameras that created them. Starting in *PRMan* version 11, a projection matrix is automatically stored within a tiff file that corresponds to the camera that rendered it. Using the *PRMan* built-in function textureinfo() to access a rendered image’s projection matrix, it becomes trivial to project an image from the coordinate system of the camera that created it. Below is a shader fragment demonstrating how to input a refraction image, pre-rendered from a refraction camera, into a shader during the final render pass (the same code works for reflection, or any arbitrary camera for that matter).

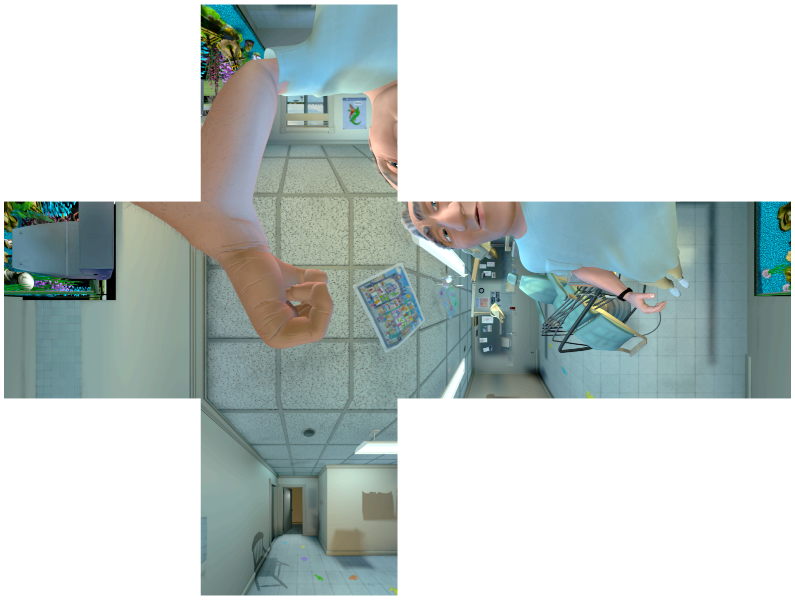
```
/* Extract the projection matrix from a refraction image */
matrix RefrMatrix;
textureinfo(“refr.tif”, “projectionmatrix”, RefrMatrix);

/* Transform P into “refraction camera projection space” */
point refrP;
refrP = transform(RefrMatrix, P);

/* Transform refrP from –1=>0=>1 space to 0=>1 space */
vector scl = vector (0.5, -0.5, 1);
vector off = vector (0.5, 0.5, 0);
refrP = refrP * scl + off;

/* Extract texture coordinates from refrP */
float ss = xcomp(refrP);
float tt = ycomp(refrP);

/* Access refraction color at current point in shader */
color Crefr = texture(“refr.tif”, ss, tt, ...);
```

In the case of reflection and refraction, the colors retrieved from the pre-rendered images should be added to `Ci` *after* your illumination calculations to avoid double illumination. Be sure to turn off the wall of glass you’re creating reflections and refractions for during the reflection/refraction pre-renders, otherwise you’ll see the glass in its own reflection/refraction in the final render (and besides, your shader will probably fail looking for the very texture that you’re currently rendering).



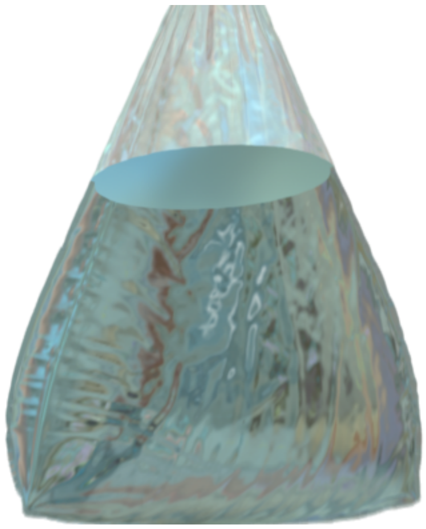
Early test using reflection & refraction multi-pass rendering

The water bag shader in “Finding Nemo” built upon the multi-pass rendering technique developed for the fish tank. The first render pass is an environment cube centered on the bag, capturing everything in the scene except the bag and its contents (like a fish).



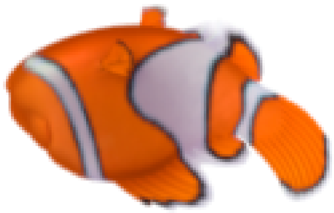
Pass 1: Environment cube of the Dentist & surrounding office

The second render pass cuts the bag in half and throws away the front, leaving only the inside back half of the bag (the back half of the bag as seen from inside the bag).



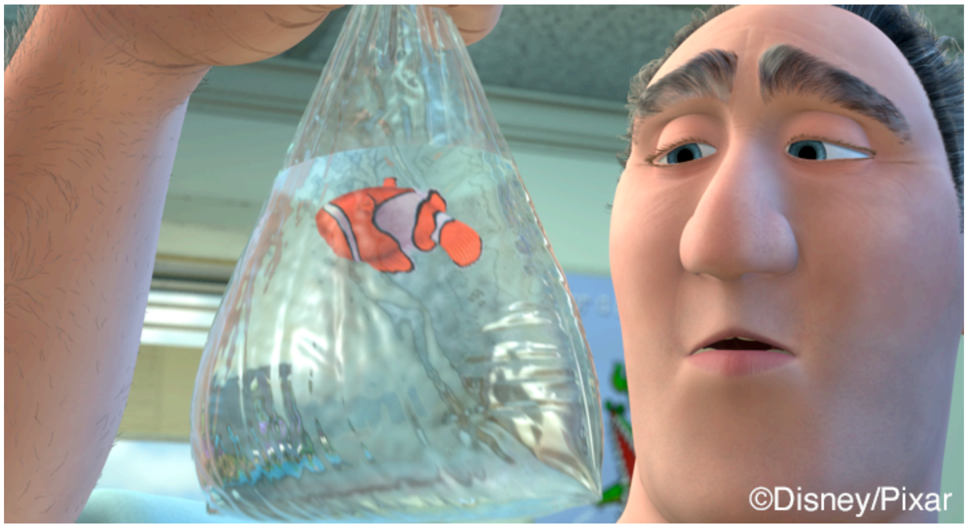
Pass 2: Back half of the bag (the camera is effectively inside the bag)

This backside render uses only the previously rendered environment cube to determine its color. The bag shader accesses the environment cube using warped surface normals to achieve a reflected and refracted appearance, and `Oi` is set to (1,1,1) so that transparency doesn't interfere with the refracted image. The next render pass consists of only the contents of the bag (Nemo), rendered onto a flat piece of geometry.



Pass 3: Nemo is rendered onto a plane in order to simulate ray-tracing

Nemo is rendered onto a plane so that the next and final render pass, the front of the bag, can simulate ray-tracing and refract Nemo accordingly. To learn more about simulated ray-tracing, refer to “RenderMan Tricks Everyone Should Know” from the 1998 Advanced RenderMan Course Notes.



The final render pass of the bag and the Dentist

The final render pass uses the pre-rendered back side of the bag in order to see a refracted view of the world behind the bag. Reflection of the world bouncing off the front of the bag is calculated from the already rendered environment cube. The reflection and refraction techniques used in the bag shader are by no means physically accurate, but the goal was to fake it well enough to be believable.

Conclusion

The biggest surprise about our reflection and refraction work was how much we could get away with using only approximate solutions. This revelation allowed us to break away from complex shader writing, and instead focus on building artistic controls that the director can manipulate.

Acknowledgments

This project would not have been a success without support from Brad Andalman, John Warren, Jesse Hollander, Brad West, Robin Cooper and Laura Philips. Thank you all!

Chapter 9

RenderMan in Pixar’s Pipeline

Guido Quaroni, Pixar Animation Studios
guido@pixar.com
April, 2003

Introduction

The RenderMan Interface specification was published by Pixar at the end of the 80s and since the introduction of a compliant renderer, at the beginning of the 90s, almost every single computer generated image at Pixar has been computed by the same application, PhotoRealistic RenderMan (a.k.a. PRMan).

Today Pixar has completed the final renders of *Finding Nemo*, our fifth feature animation rendered entirely using the same RenderMan compliant renderer. The architecture is pretty much the same but the actual code has been improved and optimized over several years. This is one of the big strengths of this software, it has been tested with complex scenes on different platforms. Pixar also sells PRMan as part of the Rendering Artist Tools and this allows for even more use, testing and features.

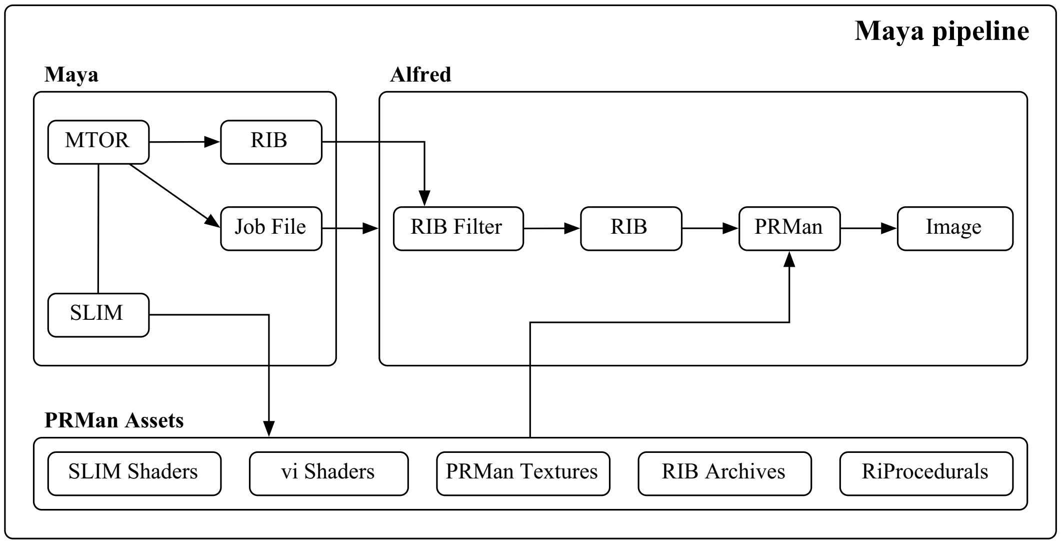
Even if the underlying code is quite complex, running PRMan is pretty simple (although there’s no guarantee it will produce a pretty picture). To be able to produce a picture, PRMan requires a RIB file, which describes a scene and access to the resources specified in the RIB file like shaders and textures. The three main environments where we access our renderer are Maya during modeling, shading and special FX, Marionette during animation, layout and lighting, Ringmaster when we need distributed rendering on the renderfarm like when computing final film frame. We also use command shell access to PRMan for debugging, fixes and where we simply have a RIB file to render.

Pixar uses several computing platforms: SGI IRIX, Sun Solaris and Red Hat Linux. While we are phasing out the IRIX OS, we are seriously looking at other attractive Unix-based alternatives. The renderer needs to run on these platforms producing the same images from the same input data. In addition binary modules used by the renderer must be compiled for the different platforms and made available to the correct architecture that is running the renderer. This has been proven to be quite a challenging task since there are subtle changes on the different architectures that must be taken into consideration. As of today we use Linux boxes at the desktop and a combination of Linux and Sun Solaris machines in the renderfarm.

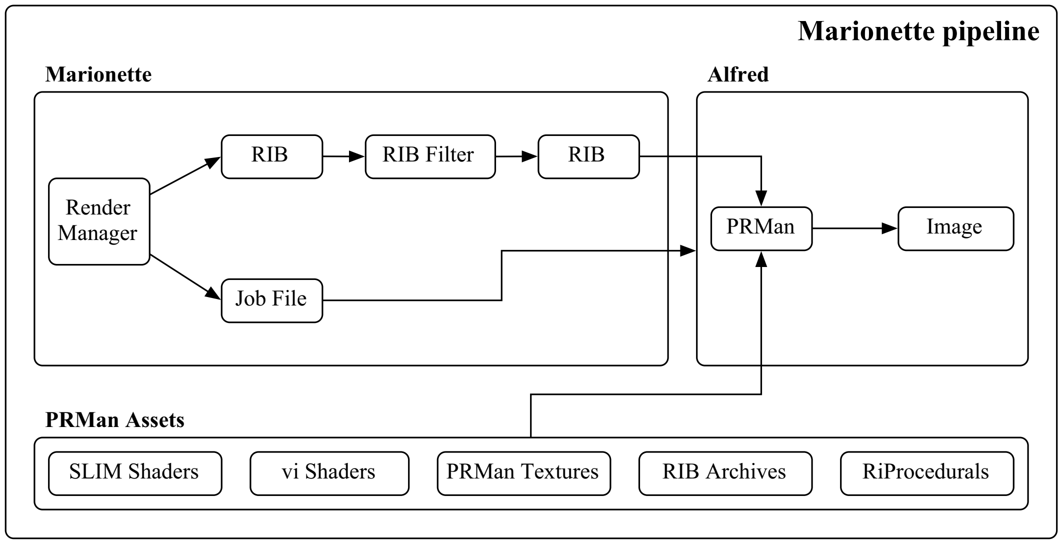
Since we always use the same renderer (or at least for 99.99% of the shots rendered since Toy Story), I assume that every mention to a renderer is actually PRMan and that every surface material or light is a RenderMan compliant shader.

Working Environments

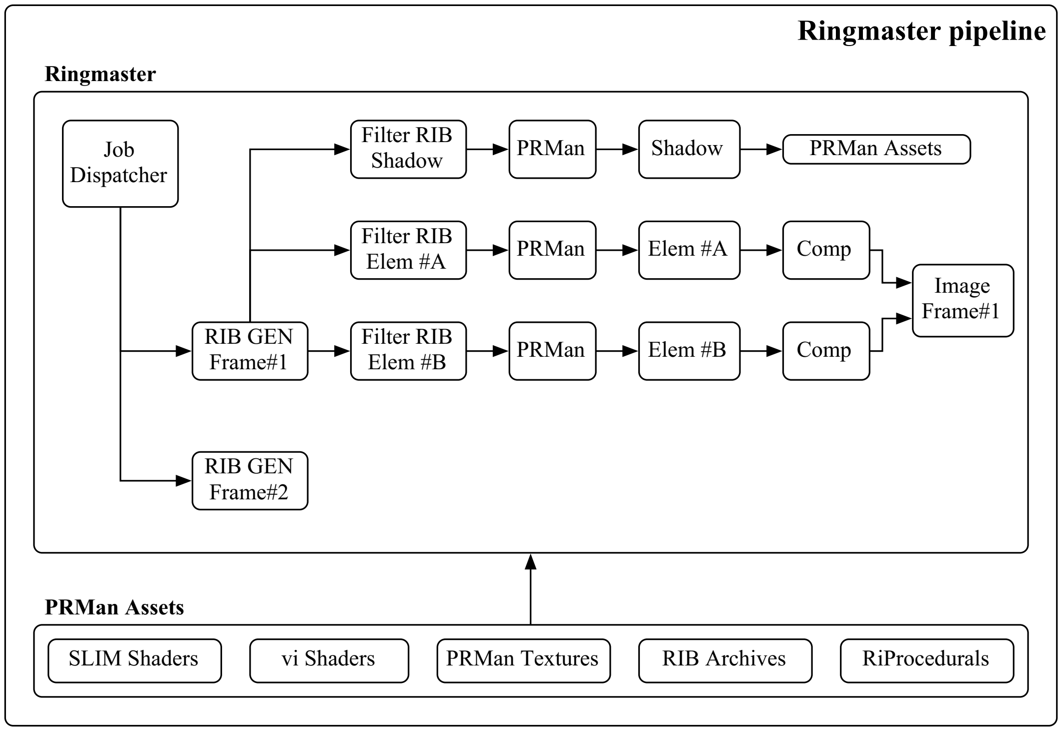
Pixar’s Artists and TDs interface with RenderMan from within 3 primary working environments: Maya, Marionette and Ringmaster. In addition to these environments, we rely on specific tools for RIB generation (MTOR), shader construction (Slim), RIB filtering (filterrib) and network rendering (Alfred). This section will describe our pipeline to RenderMan from within each environment.



In Alias|Wavefront's Maya we create RIB files using the MTOR plug-in. The file is then filtered with an internal application called `filterrib` that allows us to modify any RIB statements using custom DSO modules. For example, using `filterrib` we can add several custom RiOption and RiAttribute calls that are specific to our internal pipeline. Shaders are authored via Slim. They can be generated on the fly by Slim or they can be instances of existing ".slo" files. Using search paths, we allow MTOR renders to access our shared assets like textures, shaders and archives.



Marionette is our proprietary animation system and within it we can perform PRMan renders at a number of stages of its pipeline. The pipeline is similar to the Maya pipeline but differs in that shaders are not built at render time and that the system performs RIB filtering prior to sending the render job to Alfred.



Ringmaster is our proprietary distributed rendering system. It comprises Alfred, Pixar's job distribution product and a complex collection of propietary scripts, programs and databases. In Ringmaster we perform all the offline renders and simulations for any given shot or sequence. The key aspect of this system is the complete scriptability of the dependency between jobs and the capability to manage thousands of render jobs at any given time. For example at the peak of Nemo production we had 3000 CPUs running in our renderfarm and at the same time more than 10000 tasks were waiting to go. An interesting design aspect of this systems is that only a single RIB for a given frame is created. To render an individual element, we apply a RIB filter to accomplish per-element configurations like switching cameras, render options and attributes and object visibility.

Production Process

To better describe how RenderMan is part of the Pixar pipeline, I'll divide our production process into different categories where we take advantage of our rendering technology.

Geometric Modeling

This is the first phase of the production process that involves CG and RenderMan. As of today, we use different 3rd party modeling packages to create our models with Maya the preferred tool. During the construction and sculpting of the different surfaces (NURBS or Subdivision Surfaces) production TDs rely heavily on preview renders for different types of feedback. Here a list of useful preview renderings during geometric modeling:

Topology Checking

- Bad polygonal mesh topology won't render properly
- Simple UV shader can display NURBs orientation
- Normals properly set and consistent across several meshes

Curvature Checking

- Subdivision surface limit surface curvature only visible in a render
- Simple curvature shaders for NURBs curvature display

Surface Shading assignment
- Using special light shaders that turn unassigned objects bright red

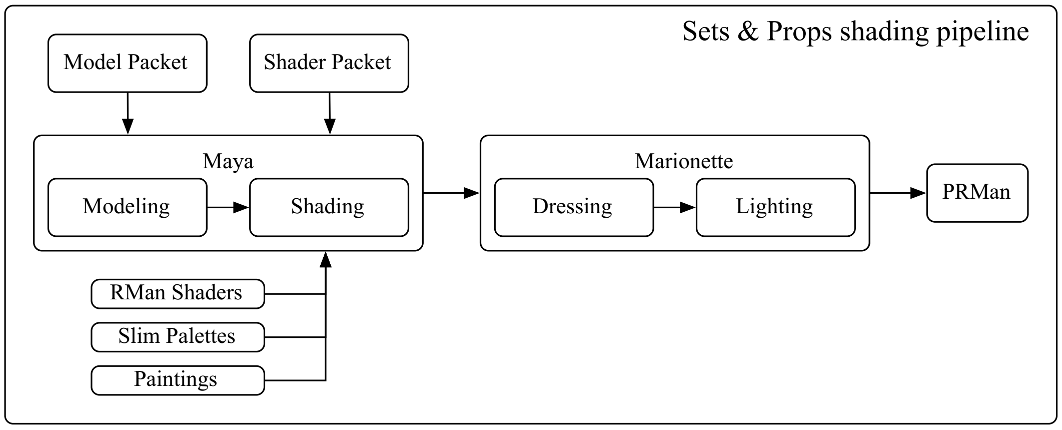
Director's Review
- Single frame or animation (turntable) for director's review

Procedural Modeling

For complex procedurals models like hair or particles, we take advantage of the RiProcedural functionality available in the renderer. RiProcedural DSOs (Dynamic Shared Objects) are modules of compiled C code linked inside the renderer at runtime. Those modules are very powerful since they allow the generation of a large number of primitives within the rendering engine context. Inside the generation context we can also take advantage of Level of Detail information like image size, camera distance and on-screen coverage of the generated primitives. The disadvantages of using DSOs are mainly in the fact that a bug in the module can crash the entire render. Also extra attention has to be paid in a multiplatform environment like our studio. We define search paths in the RIB file to be platform independent and use a combination of macro expansion (`$ARCH`) and RIB file filtering to ensure matching architecture between the renderer and the modules and allow customization and versioning.

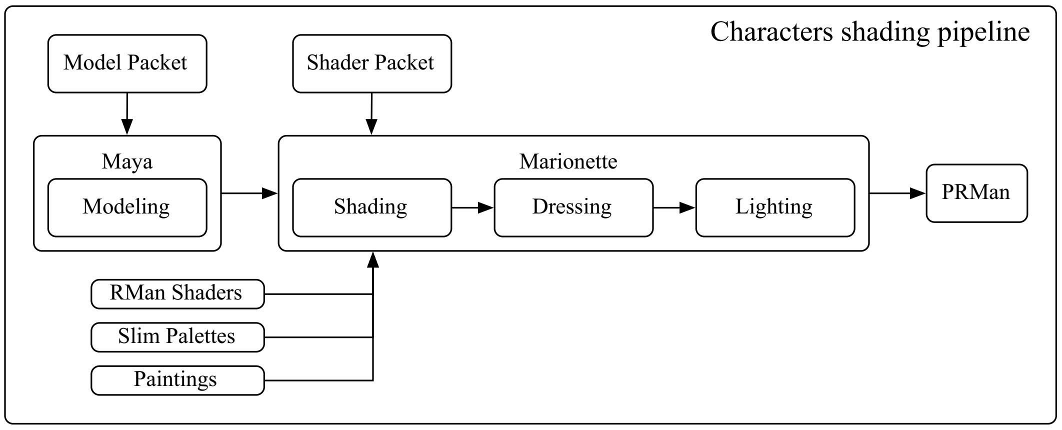
Shading

The shading process is performed in different environments. In Slim running in standalone mode, the art department defines the material palette for a given model. In general we start from a simple Slim appearance where basic properties like color, opacity, shininess, roughness, etc. are defined. Sometimes a material palette can be simply a list of material names with a single RGB color associated. The level of complexity of these materials may vary from model to model or from production to pro-



duction.
For models built in 3rd party modelers and in particular for sets and props, we use Maya for the definition of the various shaders and their binding with the objects. Here using MTOR and Slim, technical directors import material and/or color palettes defined in the art departments and start creating Slim shaders using the graphical UI. As part of the shading process, the surfaces can be parametrized with scalar fields, UV sets and reference shapes.

During the shading process the TD has full access to lighting setups from different environments where the model will be actually used. In particular cases, the lighting setup can be imported from specific shots of the movie. After the "final" from the director, the model is converted into the Pixar animation



environment, Marionette for dressing, lighting and final rendering.

Characters and other complex deformable models are shaded primarily by custom shaders create with a text editor and bound from within the Marionette environment.

Layout & Animation

Once models are built, shaded, and articulated, the production process moves entirely inside Marionette. PRMan is used by both the layout and animation department for intermediate renders of one or more shots.

For layout renders the system performs render tests of single frames and/or entire shots. The software generates the RIB file for every frame and before rendering some filtering is performed on these files to create "variants" based on user settings. For example, in layout renders we use some generic lighting environment defined by basic lights attached to the camera and we replace all the shaders with a simple surface shader. This speeds up the render time at the expense of quality. This trade off is acceptable in layout as render quality is not critical when evaluating camera angles. Once the camera is locked, a render of the background is performed to create an "image plane" with the set and props.

The animation department relies on similar rendering settings. They preview their animations using either OpenGL renders or when detail is required or the models require simulations on PRMan renders.

Special FX

Special FX are performed either in Marionette or Maya depending on the shot complexity and requirements. Particular attention has to be taken when mixing the two environments, especially when lighting has to match the two plates.

Lighting and Film Rendering

Lighting and Film Rendering are the places where all the features are turned on and we get the maximum complexity. In those departments, before submitting the final render, TDs writes complex Ringmaster scripts and rules to create all the elements with the proper dependency analysis.

Film Rendering is also the most CPU and memory intensive renders so in Ringmaster we use all sort of optimizations. For fast re-rendering on the desktop, we use a special pipeline that involve a pre-render pass that compute what we call a "deep" file and then a re-render pass where Lumiere (an internal version of Pixar's Irma re-renderer) recomputes only the changes from the pre-render pass. This is extremely useful for light tweaking and previewing. Another technique for accelerating the lighting process requires per-light rendersto feed a custom application which performs pixel-based lighting tweaks.

Conclusion

Over several years of continuous use, RenderMan has become a fundamental aspect of our production pipeline. Understanding the RenderMan specifications and being able to write shaders is a must for almost of every technical director at our studio. As we move forward with new features added to PRMan, we are now in the process of improving our pipeline to take full advantage of new capabilities like ray tracing, occlusion maps, global illumination and other new advanced features we are going to add in the not-too-distant future.

Acknowledgements

The RenderMan pipeline at Pixar is the result of several years of hard work by lots of great people at our studio, both from production and the Studio Tools department. In particular I'd like to thank Brad West, Sanjay Bakshi and Dayiv Ryu for their contributions iin improving the Pixar shading pieline in the last 18 months and to Wayne Wooten, Susan Fisher and Tony Apodaca, lead of the Film Render group, for keeping the end of the pipeline, the final renders, under control during the production of *Finding Nemo*. I'd like also to thank the engineers of the RenderMan Products group for the development of MTOR and Slim and for the amzing work they are doing on PRMan.

Shading a Coral Reef

Chris Bernardi
Pixar Animation Studios

Overview

The shading of large, organic sets for computer graphics poses several challenges to a production. “Finding Nemo” presented Pixar with the task of shading a coral reef to serve as the backdrop for a large section of film. Working from pastels and sketches provided by our art department, we were asked to create a completely invented environment yet still maintain a sense of realism. Through proper planning, and the creative use of Pixar’s PhotoRealistic RenderMan, we were able to tackle these challenges and provide a rich backdrop for our story. These notes look specifically at the process of shading the coral itself, from early shader design through final optimizations.

Preproduction: What You Don’t Know Can Hurt You

In January of 2001, we began some of the initial preproduction tests for coral shading. Because of the many technical challenges facing the production, it was decided that we would build a miniature coral reef and take it all the way through our current pipeline to final film render. As a result of this test, we were able to discover several properties that our coral shaders would need to possess.

Shader Flexibility

Shading changes for the coral are will need to be made late in the pipeline, possibly as late shot lighting. New coral looks will need to be created on short notice.

Sub-Surface Scattering

None of our current illumination models will be adequate to handle the sub-surface scattering that will be required for the coral. This look will need to be achieved with little setup time and minimal impact to render times.

Optimizations

With so much geometry in the set, optimizations will be required in both modeling and shading to achieve manageable render times.

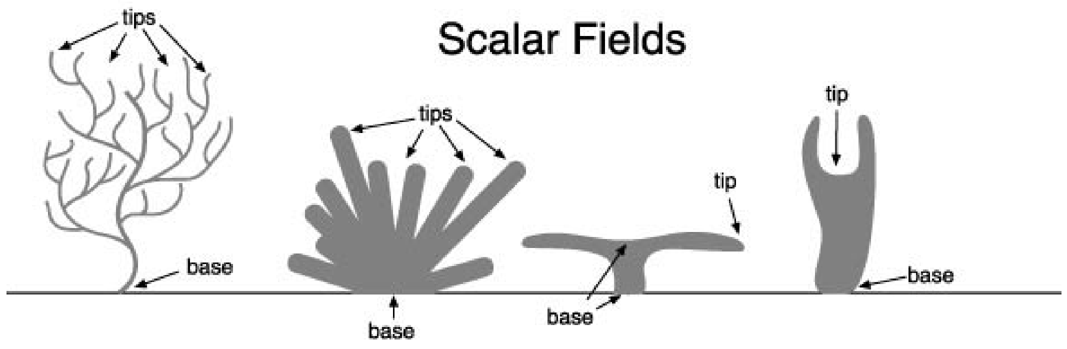
Shader Flexibility

One Shader to Rule them All

From shading the corals in preproduction, we discovered that all of the corals we needed to create had some very similar shading properties. We also found that when we split up the shader writing between too many people, we began to have problems maintaining a visual consistency on the set. This often led to corals that had been modeled and shaded going unused by the set dressing department, because they simply did not work visually. This, coupled with the need for flexibility, led us to believe that the construction of a larger shader that could handle the majority of the corals was going to be the best approach. The added benefit was that changes and optimizations could be handled from a single code base.

There has always been a continuum of approaches to shading with the “one model/one shader” approach on one end of the spectrum, and the “handful of shaders for everything” at the other end. While Pixar often tends toward the former, the coral represented a situation where a more global shading approach was practical. This is often the case in natural, organic settings (e.g. it doesn’t make sense to write a unique shader for every blade of grass in a field).

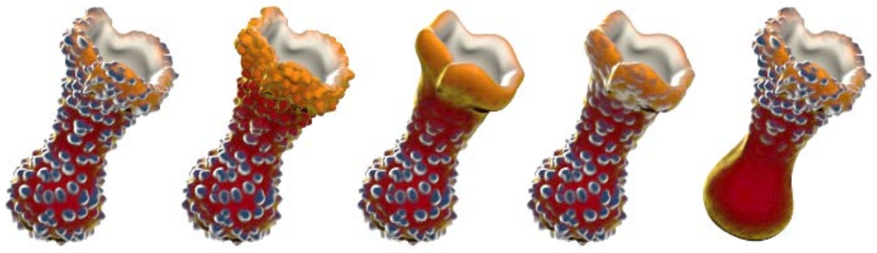
Our initial thoughts were that we might end up with 4 or 5 shaders, but they would share a common collection of shading language functions. As it turned out, all the coral in the film was shaded with a single shader. The same shader found it’s way into applications beyond the coral reef, such as rocks and even the throat of Nigel, the pelican.



Scalar Fields

We knew that we would want to control the placement of the different layers of the shader along the length of the various corals. Since many of our models were subdivision meshes, lacking in coherent texture coordinates, we chose to do this using scalar fields. A scalar field is simply a piece of information (in our case, a float value) that is associated with a vertex (or CV) and smoothly interpolated. These are also known as primVars in shading systems like Slim. They are passed into the shader as a varying float. You can use this much like a st coordinate system, but with only a single value associated with each vertex. The information we wanted for each vertex was the distance along the surface to the base of the coral, and the distance to the tip of the coral. Note that one is not necessarily the inverse of the other, given the branching structure of many corals.

For corals that were generated procedurally (such as the big plate corals), the scalar field assignment was built into the procedural code. Models built in Maya required a little more work. While we had considered other techniques such as unwrapping the models or painting the values with Artisan, we chose instead to develop a fairly simple script in MEL. The technical director needed only to select the vertices that comprised the tips (or base) of the coral and run the script. The script would "walk" across the subdivision mesh out from those points, dropping down increasing values into the scalar field. It would then go back and re-normalize the values into a 0-1 range. While the result was not a direct



Polyp Suppression Using Scalar Fields

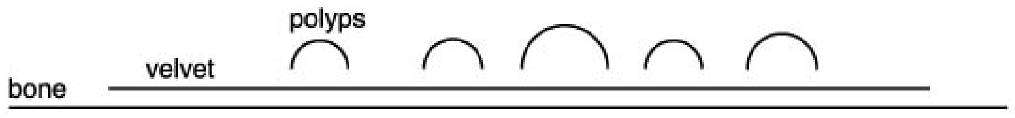
measure of the distance (it was a measure of the number of vertices traversed) the regularity of the meshes we were working with made any differences inconsequential. The result is an additional set of values available in the shader that gave us an indication of where we were along the length of the coral.

In addition to using the scalar fields for placement of shading layers, they were also used for shading tasks such as suppressing polyp sizes color transitions. They also ended up playing a critical role in the development of our sub-surface scattering look as well.

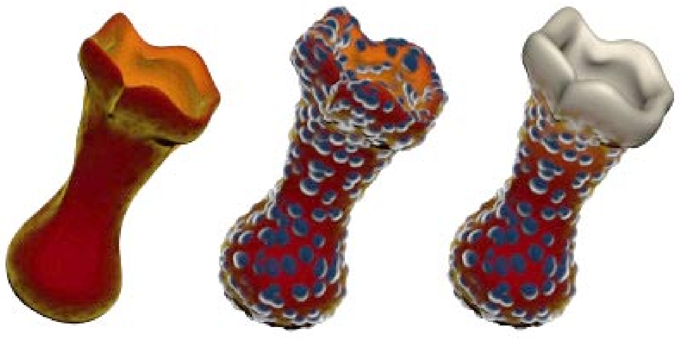
Shading in Layers

Before getting into the nuts and bolts of writing a shader. It's usually a good idea to try to break apart the visual elements of what you are trying to create. Working closely with the Art Department we were able to break the shading of the corals into three major elements, or layers.

Bone:	the bony calcium carbonate "skeleton" of the coral
Velvet:	the soft looking main surface of the coral
Polyp:	soft bulb structures that protrude out of the velvet layer



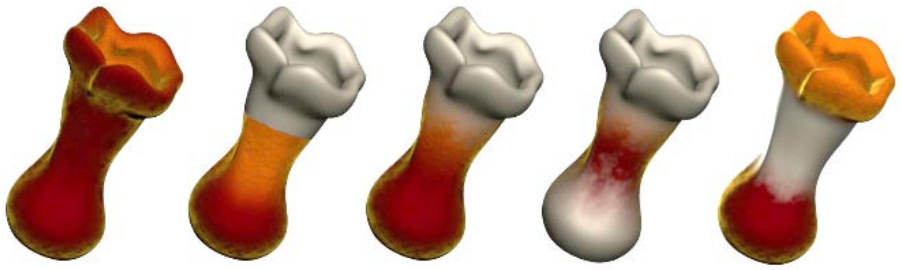
Coral Material Layers



Coral Material Layers

While these layers (and their names) are only loosely based on the physical structure of real corals, they reflect the visual features that we wished to represent. This created a common language with the Art Department when features of the coral were being discussed. This was particularly important since it was matching the Production Designer's vision that was our goal, not reproducing reality. With each layer existing in different physical locations across the coral, we were able to treat them much like independent shaders, each with its own illumination.

For both aesthetic reasons, as well as efficiency, transitions between the layers were usually abrupt. For example, we never set the velvet layer to show through all over to the underlying bone layer. This would force all of the code for each of the layers to be executed, including illumination. Rather, each layer was treated more as a discreet and opaque layer of the coral, with some blending occurring in the transition areas. The transitions themselves often made use of the underlying patterns to make the transitions more interesting. See "Appendix: More Interesting Transitions" for more details and code examples.



Coral Layer Placement

Assorted Textures used for Polyps

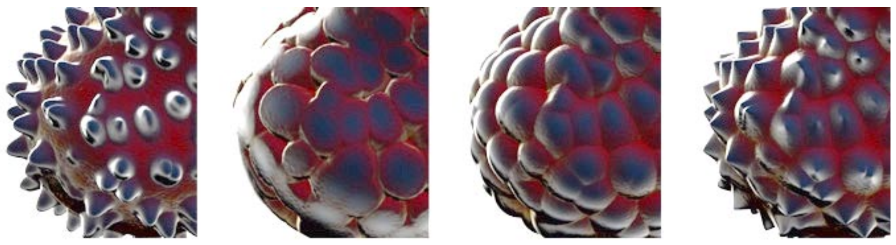
Patterns

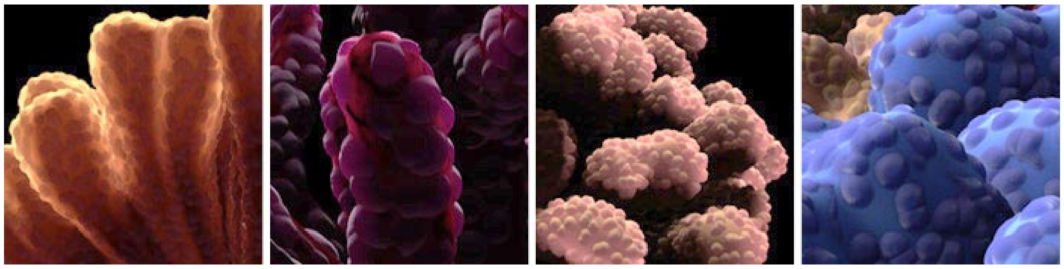
The patterning of the layers (including coloration, bump/displacement, and modulation of illumination parameters) was achieved through combinations of texture and procedural methods. 3D Painting directly on the individual corals was avoided due to the shear number of coral varieties required, as well as the complexity of setting up projections or proper parameterization for many of the complex branching corals. When we needed texture coordinates, a modified version of a cubic projection was used instead to accommodate the varied geometry.

All of the major patterns available in the various layers were set up in advance with the ability to access over 30 different texture/procedural combinations. Some the patterns were painted, some were generated procedurally in the shader, and others were initially written as RenderMan shaders and then baked out into textures for convenience and speed.

On layers such as the polyps, additional processing controls were added to allow maximum flexibility of the existing layers. This was particularly useful for displacement, where the results of the processing are directly seen as changes in geometry.

The combination of these three layers and a flexible system of pattern generation gave us exceptional control over the look of our corals, as well as a common code base and familiar set of controls across all of the corals in the reef.


Polyp Displacement Filtering



Early Scattering Tests

Sub-Surface Scattering: Well, Sort of....

We knew coming out of preproduction that some form of sub-surface scattering (we refer to this simply as scattering from here on out) was going to be required. This is the effect when light enters into the medium of an object, and is scattered instead of reflected. The reference we received from the Art Department was to hold our hand up to a light bulb and observe how the light passed through the skin. While much work has been done in this area, and several approaches were underway at Pixar at the time, the coral shading had a unique set of constraints. The scattering look needed to incur very little setup time on the part of the lighters, and the effect could not impact our render times significantly. This seemed to eliminate most approaches we had seen so far. Instead, we chose to approach the problem from a different angle. We knew that we needed to look at this not as a general sub-surface scattering solution, but as a very specific visual property that needed to work only for our limited case corals. The resulting approach, while lacking in elegance, provides insight into alternative methodologies for approaching complex visual problems. The entire process was done over the course of several days and provided a fast and efficient way to achieve the very specific look that was required.

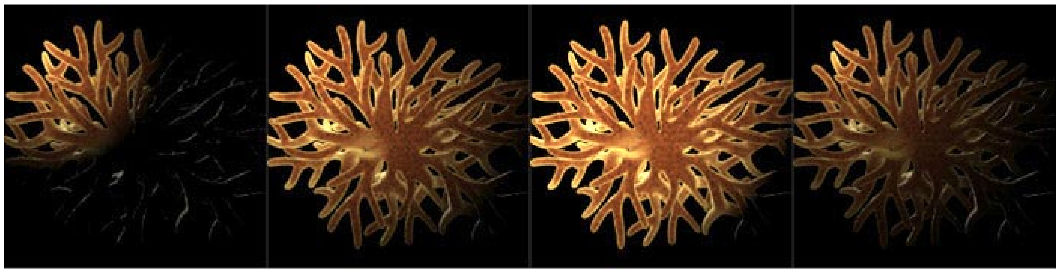
The approach was somewhat unique for computer graphics illumination, but quite familiar to anyone with a background in sound synthesis. Illumination in computer graphics is usually based on an additive approach. Starting with a black surface the various components of illumination (e.g. diffuse, specular etc.) are added up. The approach we took with coral scattering was a subtractive approach (really a multiplicative, but that's splitting hairs). The idea was to assume that every light has a 100% contribution to the entire surface of the coral, discarding the influence of view angles and surface normals. This is similar to ambient lighting techniques. For any light shining on the coral, the entire surface glows based on the intensity of the light. This was made into a simple illumination loop so we could have access to shadowing.

At this point, a mock-up was done in Photoshop to explore what terms might be useful in modulating this uniform glow. A variety of greyscale images were generated that included scalar fields, dot products of the view angle and surface normal, some texture passes and layer masks. By working in a compositing environment, we were quickly able to determine which elements might prove useful to our final illumination procedure. In the end, it turned out that the dot product of the view vector and the surface normal (basically a measure of how much the surface is facing the camera, referred to as normalDot from here on out), a texture, and a scalar field of the tip distance was all that was initially required to give us a satisfactory look. The resulting code resembled something like this:

```
Cscatter = mix(Cinner, Couter, tipDistance*normalDot);
Cscater = mix(Cscatter, texColor, blurryTexture);
Cscatter *= scatterIllum(P);
```

We used the normalDot as a naive measure of the distance the light would need to travel if it were behind the object. While this is somewhat true for surfaces like spheres, it does not hold up as well for arbitrary geometry. By multiplying the scalar field for tip distance by the normalDot, we were able to eliminate most of the "xray" effect that often results from using values like normalDot for illumination. This was consistent with the direction we were getting from the Art Department, who wanted the scattering look to appear more pronounced at the tips (or edges in the case of the plate corals). We used the result to mix an inner and outer color. The inner color tended to be more saturated and represented the color of the sub-surface media, while the outer color was less saturated and tended to take on more of the surface color. We noticed that if we calculated our normalDot prior to displacement of the polyps, we could achieve a far more realistic look. This gave the polyps that were on the silhouette edge of the object a nice translucent feel. We then introduced a texture that we had blurred to represent varying densities in the sub-surface media. While this is really a hack and does not provide any parallax or true volumetric effects, its use was subtle and provided that extra level of detail that was needed. The textures were generated in layers with varying amounts of blur and transparency to fake a sense of depth. The final shader code offered controls over the influence of these various components into the final scattering. We then factored the result into the new illumination loop we had constructed.

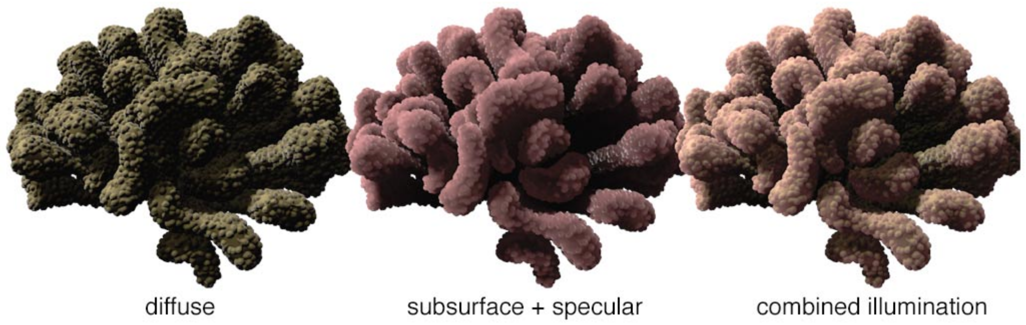
The initial results were quite promising. The test results look very nice when the light was behind an object, but not when the light was in front. While our lighters had the ability to turn off this illumination on a per-light basis, we decided to add a slight improvement to the illumination to offer more control from within the shader.
We simply added a ramp to the effect, based on how much the light was pointing at the camera versus away from the camera.



Scattering Falloff

This not only solved our initial problem, but gave the overall illumination a much more natural feel. The position and angle of the light had a much more realistic affect on the final look. The result was the creation of parameters to control the tendency of light to scatter forward only, versus scattering in all directions. A series of animations with the camera orbiting some typical coral was used to balance the transition from diffuse to scattering illumination.

A quick glance at the approach reveals that shadows could cause significant problems to our illumination model. If the lighting has it's greatest influence when it is directly behind an object, then a shadow from that light would essentially remove all the illumination we have created. We had developed several schemes for handling this (such as reversing normals during shadow renders to cull the front faces). After several tests with some of the lighting crew, we discovered that with judicious use of shadow bias, shadow blur and shadow density, we actually preferred the look as it was.



Combining Diffuse and Scattering Illuminations

Variation

Whenever you apply the exact same shader to a collection of geometry, you run the risk of all the models looking the same. This is particularly true if you are applying the shader to a limited set of models, as we were in the coral reef. Much of this can be eliminated by minor adjustments to the parameters of the shader for each model. This can prove to be a time consuming task when you have thousands of models in a scene. A simple technique was used to add variation to the corals and help break up these similarities.

This approach begins by passing a unique integer to each of the models. This integer is then passed to a call to cellnoise() in the shader, resulting in each model having a unique float associated with it. This vector was then used to offset shader space, offset the colors (often in HSV space) and offset the overall size of the features. While the results can be subtle, the absence of these variations led to objectionable similarities among the coral.

Optimization

With the large amount of geometry we planned on using in the reef, we knew that optimization was going to play an important role in managing our rendering times. Aside from standard code optimizations, we made extensive reuse of function calls and variables. Single channel textures of reasonable sizes (usually 1k) were used whenever possible, with color information introduced in the shader. Information flow was kept as float data until it was absolutely needed as color information. Calls to noise (particularly fractal noise) were kept to a minimum and was often reused in various aspects of the shading. It was often found that a texture was always being run through a series of processing functions before it was used (smoothstep, pow etc.). When this occurred, a new texture was generated with these functions baked in and the modifying functions turned off.

When working with large shaders that are going to be used for a variety of purposes, it is important to be able to control what aspects of the shader are executed for any given model. The shader was written in a very modular fashion, bounded by conditional statements. The logic for these conditionals was based in the model. This allowed the model to execute the conditionals once per frame, and shut down large blocks of the shader (often entire layers) by passing a uniform parameter to the shader. For example, if the displacement parameters for the polyps were set to zero, then ignore all code (texture calls, modifiers, modifications of P) that are associated with that displacement.

The result of these optimizations gave us an order of magnitude variation in render time for any given coral, based entirely on the complexity of the shading parameters. As a general practice, models were initially set with parameters that would execute quickly and only enhanced if they were required. While we had considered a more automatic method, this approach did not require very much time and gave us the ability to make rendering speed decisions based on all the aesthetic elements of a scene (including depth-of-field and fog).

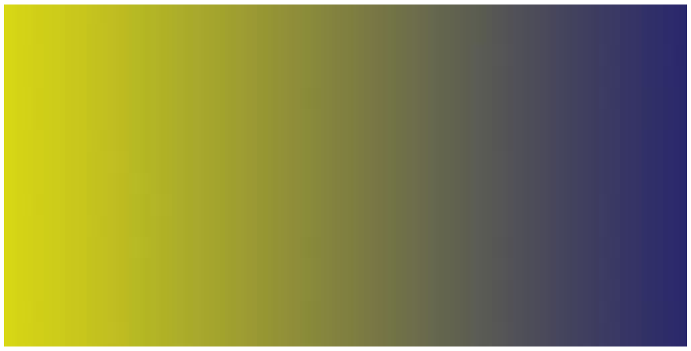
While we had considered using additional optimizations, such as level-of-detail and cards, we were pleased enough with the render times that the additional work was not merited.

Conclusions

Large global shaders can, and should, play a specialized role in the shading pipeline of any large production. The success of the coral shader led to its application to a variety of surfaces that extended far beyond its original purpose. By focusing on its practicality and impact on lighting, no special setup was required to achieve the backlighting effects. The lighting lead for the coral reef commented "I just point some lights at the coral and they look great." This is music to the ears of anyone who has had to do last minute fixes to shaders once they have entered into the hectic shot lighting phase of production.

Appendix: More Interesting Transitions

When you are creating a mask to blend or mix two elements in the shader, it almost always helps to break up the transition. For example:



```
uniform color Cyello = color (1, 1, 0);
uniform color Cblue = color (0, 0, .5);
float ramp = smoothstep (.25, .75, s);
color Cfinal = mix(Cyello, Cblue, ramp);
```

This is not very interesting. We'd really like to break this up in some fashion. While there are any number of ways one might go about this, here is a fairly simple one.

```
uniform float distort = .25;
uniform color Cyello = color (1, 1, 0);
uniform color Cblue = color (0, 0, .5);
/*texture noise or any interesting pattern*/
float pattern = noise(s*20,t*20);
float ramp = smoothstep (.25, .75, s + (pattern -.5)*distort);
color Cfinal = mix(Cyello, Cblue, ramp);
```

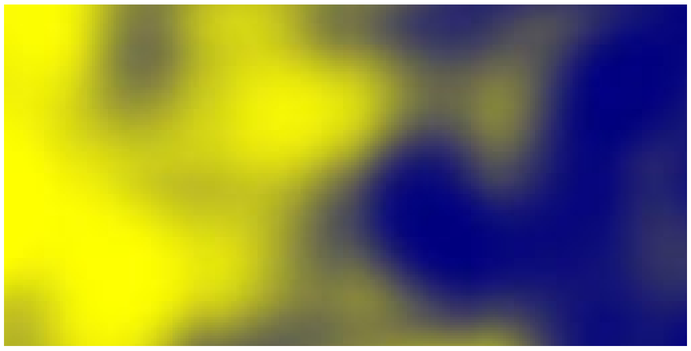
Here we have offset the s component of the texture space in which we are calling the ramp by some interesting pattern.

Another approach involves changing the way you think about the smoothstep function. Rather than pass a high and low point for the smoothstep, you can write a little function to pass a center and a blur. This will function more like a step or threshold function, but with a variable blur component.

```
float blurstep (float center; float blur; float pattern){
return = smoothstep(center-blur/2,center+blur/2,pattern);
}
```

Using this new function, we can rewrite our transition in a more powerful way

```
uniform float blur = .2;
uniform color Cyello = color (1,1,0);
uniform color Cblue = color (0,0,.5);
/*texture noise or any interesting pattern*/
float pattern = noise(s*20,t*20);
float ramp = blurstep(pattern, blur, smoothstep(.25, .75, s));
color Cfinal = mix(Cyello, Cblue, ramp);
```



The result is still a function that runs from 0 to 1 across s (well, basically) but we now have a far more interesting pattern, as well as the ability to control the blur.

Don’t forget that if you aready have an interesting pattern variable that you are using somewhere else in your code, you might be able to reuse it here and pick up a lot of organic complexity at very little rendering cost.

Reprints

This reprint originally appeared as:

Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy and Pat Hanrahan. A practical model for subsurface light transport. In *Proceeding of ACM SIGGRAPH 2001*, pages 511-518, August 2001.

A Practical Model for Subsurface Light Transport

Henrik Wann Jensen Stephen R. Marschner Marc Levoy Pat Hanrahan

Stanford University

Abstract

This paper introduces a simple model for subsurface light transport in translucent materials. The model enables efficient simulation of effects that BRDF models cannot capture, such as color bleeding within materials and diffusion of light across shadow boundaries. The technique is efficient even for anisotropic, highly scattering media that are expensive to simulate using existing methods. The model combines an exact solution for single scattering with a dipole point source diffusion approximation for multiple scattering. We also have designed a new, rapid image-based measurement technique for determining the optical properties of translucent materials. We validate the model by comparing predicted and measured values and show how the technique can be used to recover the optical properties of a variety of materials, including milk, marble, and skin. Finally, we describe sampling techniques that allow the model to be used within a conventional ray tracer.

Keywords: Subsurface scattering, BSSRDF, reflection models, light transport, diffusion theory, realistic image synthesis

1 Introduction

Accurately modeling the scattering of light by materials is fundamental for realistic image synthesis. Even the most sophisticated light transport algorithms fail to produce convincing results if the local scattering models are too simple. Therefore a great deal of research has gone into describing the scattering of light from materials.

Previous research has focused on developing models for the bidirectional reflectance distribution function (BRDF). The BRDF was introduced by Nicodemus [14] as a simplification of the more general bidirectional surface scattering distribution function (BSSRDF). The BSSRDF can describe light transport between any two rays that hit a surface, whereas the BRDF assumes that light entering a material leaves the material at the same position (Figure 1). This approximation is valid for metals, but it fails for translucent materials, which exhibit significant transport below the surface. Even for many materials that do not seem very translucent, using the BRDF creates a hard, distinctly computer-generated appearance because it does not locally blend surface features such as color and geometry. Only methods that consider subsurface scattering can capture the true appearance of translucent materials, such as marble, cloth, paper, skin, milk, cheese, bread, meat, fruits, plants, fish, ocean water, snow, etc.

1.1 Previous Work

Almost all BRDF models are derived exclusively from surface scattering, with any subsurface scattering approximated by a Lambertian component. An exception is the model by Hanrahan and Krueger [10] which includes an analytic expression for single scattering in a homogeneous, uniformly lit slab. However, all BRDF models ultimately assume that light scatters at one surface point and they do not model subsurface transport from one point to another.

Subsurface transport can be simulated accurately but slowly by solving the full radiative transfer equation [1]. Only a few papers in graphics have taken this approach to subsurface scattering. Dorsey et al. [5] simulated full subsurface scattering using photon mapping to capture the appearance of weathering in stone. Pharr and Hanrahan [15] used scattering functions to simulate subsurface scattering. These approaches, while capable of simulating all of the effects of subsurface scattering, are computationally very expensive compared to the simulation of opaque materials. Techniques based on path sampling are particularly inefficient for highly scattering materials, such as milk and skin, in which light scatters multiple (often several hundred) times before exiting the material. For highly scattering media Stam [17] introduced the use of diffusion theory. He solved a diffusion equation approximation using a multigrid method, and used this method to render clouds with multiple scattering.

Subsurface scattering is also important in medical physics, where models have been developed to describe the scattering of laser light in human tissue [6, 8]. In that context, diffusion theory is often used to predict as well as to measure the optical properties of highly scattering materials. We have extended this theory for use in computer graphics by adding exact single scattering, support for arbitrary geometry, and a practical sampling technique for rendering.

In measurements of appearance for computer graphics, subsurface scattering has rarely been considered. Debevec et al. [3] measured light reflection from human faces, which included contributions from subsurface scattering, but they did not relate the data to the physical properties of the material. Again building on medical physics research [8, 9], we have extended a methodology developed for measuring biological tissues into a rapid image-based appearance measurement technique for translucent materials. This method examines the radial reflectance profile resulting from a beam illuminating the sample material. By fitting an expression derived from diffusion theory it is possible to estimate the absorption and scattering properties of the material.

2 Theory

The BSSRDF, S, relates the outgoing radiance, $L_o(x_o, \vec{\omega}_o)$ at the point x_o in direction $\vec{\omega}_o$, to the incident flux, $\Phi_i(x_i, \vec{\omega}_i)$ at the point x_i from direction $\vec{\omega}_i$ [14]:

$$dL_o(x_o, \vec{\omega}_o) = S(x_i, \vec{\omega}_i; x_o, \vec{\omega}_o)\, d\Phi_i(x_i, \vec{\omega}_i).$$

The BRDF is an approximation of the BSSRDF for which it is assumed that light enters and leaves at the same point (i.e., $x_o = x_i$). Given a BSSRDF, the outgoing radiance is computed

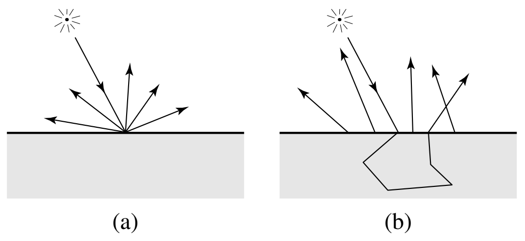


Figure 1: Scattering of light in (a) a BRDF, and (b) a BSSRDF.

by integrating the incident radiance over incoming directions *and area*, A:

$$L_o(x_o,\vec{\omega}_o)=\int_A\int_{2\pi} S(x_i,\vec{\omega}_i;x_o,\vec{\omega}_o)\,L_i(x_i,\vec{\omega}_i)\,(\vec{n}\cdot\vec{\omega}_i)\,d\omega_i dA(x_i).$$

Light propagation in a participating medium is described by the radiative transport equation, often referred to in computer graphics as the volume rendering equation:

$$(\vec{\omega}\cdot\vec{\nabla})L(x,\vec{\omega})= -\sigma_t L(x,\vec{\omega})+\sigma_s\int_{4\pi} p(\vec{\omega},\vec{\omega}')L(x,\vec{\omega}')\,d\omega'+Q(x,\vec{\omega}).$$

In this equation, the properties of the medium are described by the absorption coefficient σ_a, the scattering coefficient σ_s, and the phase function $p(\vec{\omega},\vec{\omega}')$. The extinction coefficient σ_t is defined as, $\sigma_t = \sigma_a + \sigma_s$. We assume the phase function is normalized, $\int_{4\pi} p(\vec{\omega},\vec{\omega}')\,d\omega' = 1$ and is a function only of the phase angle, $p(\vec{\omega},\vec{\omega}') = p(\vec{\omega}\cdot\vec{\omega}')$. The mean cosine, g, of the scattering angle is

$$g = \int_{4\pi} (\vec{\omega}\cdot\vec{\omega}')p(\vec{\omega}\cdot\vec{\omega}')\,d\omega'.$$

If g is positive, the phase function is predominantly forward scattering; if g is negative, backward scattering dominates. A constant phase function results in isotropic scattering ($g = 0$).

For an infinitesimal beam entering a homogeneous medium, the incoming radiance will decrease exponentially with distance s. This is referred to as the *reduced intensity*:

$$L_{ri}(x_i + s\vec{\omega}_i,\vec{\omega}_i) = e^{-\sigma_t s} L_i(x_i,\vec{\omega}_i).$$

The first-order scattering of the reduced intensity, L_{ri}, may be treated as a volumetric source:

$$Q(x,\vec{\omega}) = \sigma_s \int_{4\pi} p(\vec{\omega}',\vec{\omega})L_{ri}(x,\vec{\omega}')\,d\omega'.$$

To gain insight into the volumetric behavior of light propagation, it is useful to integrate the radiative transport equation over all directions $\vec{\omega}$ at a point x which yields

$$\vec{\nabla}\cdot\vec{E}(x) = -\sigma_a\phi(x) + Q_0(x). \quad (1)$$

This equation relates the scalar irradiance, or fluence, $\phi(x) = \int_{4\pi} L(x,\vec{\omega})\,d\omega$, and the vector irradiance, $\vec{E}(x) = \int_{4\pi} L(x,\vec{\omega})\vec{\omega}\,d\omega$. In the absence of loss due to absorption or gain due to a volumetric light source ($Q_0 = 0$), the divergence of the vector irradiance equals zero. In this equation, we introduce a 0th-order source term, Q_0, and later we will need the 1st-order source term, $\vec{Q}_1$, where

$$Q_0(x) = \int_{4\pi} Q(x,\vec{\omega})\,d\omega, \quad \vec{Q}_1(x) = \int_{4\pi} Q(x,\vec{\omega})\vec{\omega}\,d\omega.$$

S	BSSRDF
R_d	Diffuse BSSRDF
F_r	Fresnel reflectance
F_t	Fresnel transmittance
F_{dr}	Diffuse Fresnel reflectance
$\vec{E}$	Vector irradiance
ϕ	Radiant fluence
σ_a	Absorption coefficient
σ_s	Scattering coefficient
σ_t	Extinction coefficient
σ_t'	Reduced extinction coefficient
σ_{tr}	Effective extinction coefficient
D	Diffusion constant
α	Albedo
p	Phase function
η	Relative index of refraction
g	Mean cosine of the scattering angle
Q	Volume source distribution
Q_0	0th-order source distribution
$\vec{Q}_1$	1st-order source distribution

Figure 2: Selected symbols.

2.1 The Diffusion Approximation

The diffusion approximation is based on the observation that the light distribution in highly scattering media tends to become isotropic. This is true even if the initial light source distribution and the phase function are highly anisotropic. Each scattering event blurs the light distribution, and as a result the light distribution tends toward uniformity as the number of scattering events increases.

In this situation, the radiance may be approximated by a two-term expansion involving the radiant fluence and the vector irradiance:

$$L(x,\vec{\omega}) = \frac{1}{4\pi}\phi(x) + \frac{3}{4\pi}\vec{\omega}\cdot\vec{E}(x).$$

The constants are determined by the definitions of fluence and vector irradiance.

The diffusion equation follows from this approximation. Specifically, we substitute this two-term expansion of the radiance into the radiative transport equation and then integrate over $\vec{\omega}$; for the algebraic details consult Ishimaru [12]. The result is

$$\vec{\nabla}\phi(x) = -3\sigma_t'\vec{E}(x) + \vec{Q}_1(x). \quad (2)$$

Here we have used the reduced extinction coefficient, σ_t', which is given by

$$\sigma_t' = \sigma_s' + \sigma_a \quad \text{where} \quad \sigma_s' = \sigma_s(1-g)\ .$$

The reduced scattering coefficient σ_s' scales the original scattering coefficient by a factor of $(1-g)$. Intuitively, once light becomes isotropic, only backward scattering terms change the net flux; forward scattering is indistinguishable from no scattering.

In the case where there are no sources, or where the sources are isotropic, $\vec{Q}_1$ vanishes from Equation 2. Then the vector irradiance is the gradient of the scalar fluence,

$$\vec{E}(x) = -D\vec{\nabla}\phi(x).$$

Here $D = \frac{1}{3\sigma_t'}$ is the diffusion constant. This equation makes precise the intuitive notion that there is net energy flow (i.e., non-zero vector irradiance) from regions of high energy density (high fluence) to regions of low energy density.

Finally, substituting Equation 2 into Equation 1, we arrive at the classic diffusion equation

$$D\nabla^2\phi(x) = \sigma_a\phi(x) - Q_0(x) + 3D\vec{\nabla}\cdot\vec{Q}_1(x).$$

The diffusion equation has a simple solution in the case of a single isotropic point light source in an infinite medium.

$$\phi(x) = \frac{\Phi}{4\pi D} \frac{e^{-\sigma_{tr} r(x)}}{r(x)},$$

where Φ is the power of the point light source, r is the distance to the location of the point source, and $\sigma_{tr} = \sqrt{3\sigma_a \sigma_t'}$ is the effective transport coefficient. The point source results in an energy density in the volume with an exponential falloff.

In the case of a scattering medium in a finite region of space, the diffusion equation must be solved subject to the appropriate boundary conditions. The boundary condition is that the net inward diffuse flux is zero at each point, x_s, on the surface

$$\int_{2\pi_-} L(x_s, \vec{\omega})(\vec{\omega} \cdot \vec{n}(x_s))\, d\omega = 0.$$

Here, $2\pi_-$ denotes integration over the hemisphere of inward directions. Using the two-term expansion, the boundary condition is

$$\phi(x_s) - 2D(\vec{n} \cdot \vec{\nabla})\phi(x_s) = 0. \tag{3}$$

The minus sign in the second term results from the convention that the surface normal points outward, whereas the integral is over inward directions.

Equation 3 covers the case where the two layers have matching indices of refraction, but another important case is where these indices differ. When an interface exists between media with different refractive indices, there is a reflection at the interface. Assuming F_r is the Fresnel formula for the reflectance at a dielectric interface, the average diffuse Fresnel reflectance is

$$F_{dr} = \int_{2\pi} F_r(\eta, \vec{n} \cdot \vec{\omega}')(\vec{n} \cdot \vec{\omega}')\, d\omega',$$

where η is the relative index of refraction of the medium with the reflected ray to the other medium. F_{dr} may be computed analytically from the Fresnel formula [13]. However, we will use a rational approximation of the measured diffuse reflectance [7]:

$$F_{dr} = -\frac{1.440}{\eta^2} + \frac{0.710}{\eta} + 0.668 + 0.0636\eta.$$

The resulting boundary condition between two media with different indices of refraction is

$$\int_{2\pi_-} L(x, \vec{\omega})(\vec{\omega} \cdot \vec{n}_-) d\omega = F_{dr} \int_{2\pi_+} L(x, \vec{\omega})(\vec{\omega} \cdot \vec{n}_+)\, d\omega.$$

Here the $+$ and $-$ subscript means outward and inward directions respectively. This yields

$$\phi(x_s) - 2D(\vec{n} \cdot \vec{\nabla})\phi(x_s) = F_{dr}\left[\phi(x_s) + 2D(\vec{n} \cdot \vec{\nabla})\phi(x_s)\right].$$

Note that the difference in signs between the two sides of this equation occurs because one integral is over outward directions and the other is over inward directions. Rearranging terms,

$$\phi(x_s) - 2AD(\vec{n} \cdot \vec{\nabla})\phi(x_s) = 0.$$

This boundary condition is the same as when the indices of refraction match (Equation 3); the only difference is that $2D$ is replaced by $2AD$, where

$$A = \frac{1 + F_{dr}}{1 - F_{dr}}.$$

Finally, the boundary condition allows us to compute the diffuse BSSRDF, R_d. R_d is equal to the radiant exitance divided by the incident flux. The radiant exitance leaving the surface $(\vec{n} \cdot \vec{E}(x_s))$ is equal to the gradient of the fluence at the surface

$$R_d(r) = -D \frac{(\vec{n} \cdot \vec{\nabla}\phi)(x_s)}{d\Phi_i(x_i)},$$

where $r = ||x_s - x_i||$.

In the case of finite media, the diffusion equation does not in general have an analytical solution. In this paper we are interested in subsurface reflection, which is often modeled as a semi-infinite plane-parallel medium. Several authors have analyzed the plane-parallel problem for simple source geometries, in particular, approximations of a cylindrical beam entering the media. Exact formulas exist, but they involve an infinite sum of Bessel functions [9, 16]. We seek a simple formula suitable for modeling subsurface reflection that does not involve infinite sums or numerical solution of a partial differential equation.

Eason [6] and Farrell et al. [8] have developed a method for approximating the volumetric source distribution using two point sources; that is, a dipole. Eason introduced this idea and derived explicit formulae for the dipoles for various source geometries, such as a cylindrical beam, by expanding the source distributions in terms of their moments. Farrell et al. proposed using a single dipole to represent the incident source distribution. They found a single dipole to be as accurate as, or, in some cases, more accurate than using the diffusion approximation with the true source distribution.

The dipole method consists of positioning two point sources near the surface in such a way as to satisfy the required boundary condition [6] (see Figure 3). One point source, the positive real light source, is located at the distance z_r beneath the surface, and the other, the negative virtual light source, is located above the surface at a distance $z_v = z_r + 4AD$. The resulting fluence is

$$\phi(x) = \frac{\Phi}{4\pi D}\left(\frac{e^{-\sigma_{tr} d_r}}{d_r} - \frac{e^{-\sigma_{tr} d_v}}{d_v}\right),$$

where $d_r = ||x - x_r||$ is the distance from x to the real source, and $d_v = ||x - x_v||$ is the distance from x to the virtual source. Farrell et al. [8] proposed positioning the real light source at distance $z_r = 1/\sigma_t'$, or one mean free path, below the surface. They only considered light parallel to the normal. For other light directions reciprocity can be enforced by still placing the light source $1/\sigma_t'$ straight below x_i.

The diffuse reflectance due to the dipole source can now be computed.

$$\begin{aligned} R_d(r) &= -D \frac{(\vec{n} \cdot \vec{\nabla}\phi(x_s))}{d\Phi_i} \\ &= \frac{\alpha'}{4\pi}\left[(\sigma_{tr} d_r + 1)\frac{e^{-\sigma_{tr} d_r}}{\sigma_t' d_r^3} + z_v\,(\sigma_{tr} d_v + 1)\frac{e^{-\sigma_{tr} d_v}}{\sigma_t' d_v^3}\right]. \end{aligned} \tag{4}$$

Lastly, we need to take into account the Fresnel reflection at the boundary for both the incoming light and the outgoing radiance.

$$S_d(x_i, \vec{\omega}_i; x_o, \vec{\omega}_o) = \frac{1}{\pi} F_t(\eta, \vec{\omega}_i) R_d(||x_i - x_o||) F_t(\eta, \vec{\omega}_o) \tag{5}$$

where S_d is the diffusion term of the BSSRDF. This term represents multiple scattering (one scattering event is already included in the conversion to a point source). The next section explains how to compute the contribution due to single scattering.

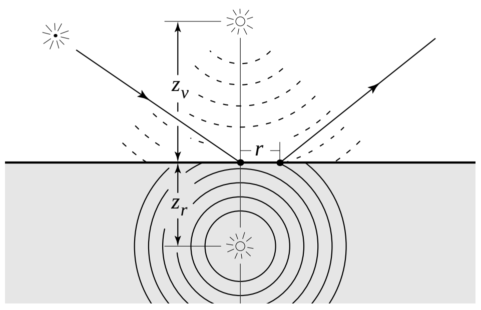


Figure 3: An incoming ray is transformed into a dipole source for the diffusion approximation.

2.2 Single Scattering Term

Hanrahan and Krueger [10] have derived a BRDF model for subsurface reflection that analytically computes the total first-order scattering from a flat, uniformly lit, homogeneous slab. In this section, we show how their BRDF can be extended to a BSSRDF in order to account for local variations in lighting over the surface.

The total outgoing radiance, $L_o^{(1)}$, due to single scattering is computed by integrating the incident radiance along the refracted outgoing ray (see Figure 4):

$$L_o^{(1)}(x_o, \vec{\omega}_o) = \sigma_s(x_o) \int_{2\pi} F\, p(\vec{\omega}_i' \cdot \vec{\omega}_o') \int_0^\infty e^{-\sigma_{tc} s} L_i(x_i, \vec{\omega}_i)\, ds\, d\vec{\omega}_i \quad (6)$$

$$= \int_A \int_{2\pi} S^{(1)}(x_i, \vec{\omega}_i; x_o, \vec{\omega}_o)\, L_i(x_i, \vec{\omega}_i)\, (\vec{n} \cdot \vec{\omega}_i)\, d\omega_i dA(x_i).$$

Here $F = F_t(\eta, \vec{\omega}_o) F_t(\eta, \vec{\omega}_i)$ is the product of the two Fresnel transmission terms, and $\vec{\omega}_i'$ and $\vec{\omega}_o'$ are the refracted incoming and outgoing directions. The combined extinction coefficient σ_{tc} is given by $\sigma_{tc} = \sigma_t(x_o) + G\sigma_t(x_i)$, where G is a geometry factor; for a flat surface $G = \frac{|\vec{n}_i \cdot \vec{\omega}_o'|}{|\vec{n}_i \cdot \vec{\omega}_i'|}$. The single scattering BSSRDF, $S^{(1)}$, is defined implicitly by the second line of this equation. Note that there is a change of variables between the first line, which integrates only over the configurations where the two refracted rays intersect, and the second line, which integrates over all incoming and outgoing rays. This implies that the distribution $S^{(1)}$ contains a delta function.

2.3 The BSSRDF Model

The complete BSSRDF model is a sum of the diffusion approximation and the single scattering term:

$$S(x_i, \vec{\omega}_i; x_o, \vec{\omega}_o) = S_d(x_i, \vec{\omega}_i; x_o, \vec{\omega}_o) + S^{(1)}(x_i, \vec{\omega}_i; x_o, \vec{\omega}_o)$$

Here S_d is evaluated using Equation 5 and $S^{(1)}$ is evaluated using Equation 6. The parameters for the BSSRDF are: σ_a, σ_s', η, and possibly a phase function (without a phase function the scattering can be modeled as isotropic). This model accounts for light transport between different locations on the surface, and it simulates both the directional component (due to single scattering) as well as the diffuse component (due to multiple scattering).

Finally, note the distances involved in both the single scattering term and the diffusion approximations. The average exit point is approximately one mean free path from the entry point. However, these two mean free paths have quite different length scales. In the single scattering case, the mean free path equals $1/\sigma_t$; in the diffusion case, the mean free path equals $1/\sigma_{tr}$. For translucent materials where $\sigma_a \ll \sigma_s'$ and consequently $\sigma_{tr} \ll \sigma_t$, the single scattering term decreases much faster than the diffusion term as the distance to x_o increases.

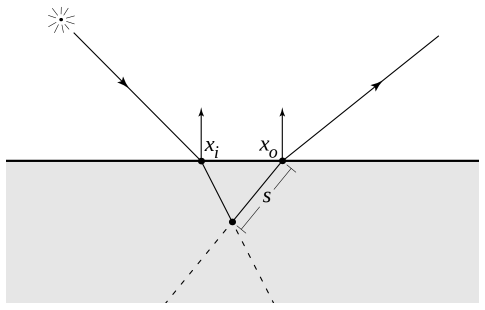


Figure 4: Single scattering occurs only when the refracted incoming and outgoing rays intersect, and is computed as an integral over path length s along the refracted outgoing ray.

2.4 BRDF Approximation

We can approximate the BSSRDF with a BRDF by assuming that the incident illumination is uniform. This assumption makes it possible to integrate the BSSRDF over the surface. By integrating the diffusion term we find the total diffuse reflectance R_d of the material as:

$$R_d = 2\pi \int_0^\infty R_d(r)\, r\, dr = \frac{\alpha'}{2} \left(1 + e^{-\frac{4}{3}A\sqrt{3(1-\alpha')}}\right) e^{-\sqrt{3(1-\alpha')}} .$$

Notice how the diffuse reflectance only depends on the reduced albedo and the internal reflection parameter A.

The integration of the single scattering term results in the model presented in [10]. For a semi-infinite medium this gives:

$$f_r^{(1)}(x, \vec{\omega}_i, \vec{\omega}_o) = \alpha F \frac{p(\vec{\omega}_i' \cdot \vec{\omega}_o')}{|\vec{n} \cdot \vec{\omega}_i'| + |\vec{n} \cdot \vec{\omega}_o'|} .$$

The complete BRDF model is the sum of the diffuse reflectance scaled by the Fresnel term and the single scattering approximation:

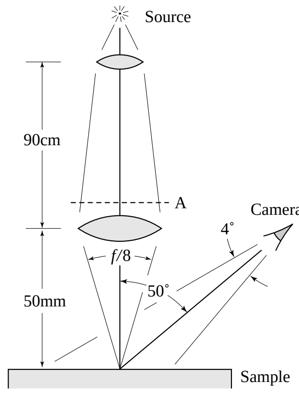
$$f_r(x, \vec{\omega}_i, \vec{\omega}_o) = f_r^{(1)}(x, \vec{\omega}_i, \vec{\omega}_o) + F\frac{R_d}{\pi} .$$

This model has the same parameters as the BSSRDF. It is similar to the BRDF model presented in [10], but with the important difference that the amount of diffusely reflected light is computed from the intrinsic material parameters. The BRDF approximation is useful for opaque materials, which have a very short mean free path.

3 Measuring the BSSRDF

To verify our BSSRDF model, and to determine appropriate parameters for rendering different kinds of materials, we used the diffusion theory of Section 2 to make measurements of subsurface scattering in several media. Our measurement approach applies to translucent materials for which $\sigma_a \ll \sigma_s$, implying that far enough away from the point of illumination, we may neglect single scattering and use the diffusion term to relate measurements to material parameters.

When multiple scattering dominates, Equation 4 predicts the radiant exitance per unit incident flux that will be observed due to a narrow incident beam, as a function of distance from the point of illumination. To make the corresponding measurement, we illuminate the surface of a sample with a tightly focused beam of white light and take a photograph using a 3-CCD video camera to observe the radiant exitance across the entire surface. We keep our observations at constant angles so that the Fresnel term remains constant for all the measurements. Figure 5(a) illustrates our measurement setup.



(a)

Material	σ_s' [mm^{-1}]			σ_a [mm^{-1}]			Diffuse Reflectance			η
	R	G	B	R	G	B	R	G	B	
Apple	2.29	2.39	1.97	0.0030	0.0034	0.046	0.85	0.84	0.53	1.3
Chicken1	0.15	0.21	0.38	0.015	0.077	0.19	0.31	0.15	0.10	1.3
Chicken2	0.19	0.25	0.32	0.018	0.088	0.20	0.32	0.16	0.10	1.3
Cream	7.38	5.47	3.15	0.0002	0.0028	0.0163	0.98	0.90	0.73	1.3
Ketchup	0.18	0.07	0.03	0.061	0.97	1.45	0.16	0.01	0.00	1.3
Marble	2.19	2.62	3.00	0.0021	0.0041	0.0071	0.83	0.79	0.75	1.5
Potato	0.68	0.70	0.55	0.0024	0.0090	0.12	0.77	0.62	0.21	1.3
Skimmilk	0.70	1.22	1.90	0.0014	0.0025	0.0142	0.81	0.81	0.69	1.3
Skin1	0.74	0.88	1.01	0.032	0.17	0.48	0.44	0.22	0.13	1.3
Skin2	1.09	1.59	1.79	0.013	0.070	0.145	0.63	0.44	0.34	1.3
Spectralon	11.6	20.4	14.9	0.00	0.00	0.00	1.00	1.00	1.00	1.3
Wholemilk	2.55	3.21	3.77	0.0011	0.0024	0.014	0.91	0.88	0.76	1.3

(b)

Figure 5: (a) Measurement apparatus, (b) measured parameters for several materials.

Because the signal falls off exponentially away from the point of illumination, the measurement must span a wide dynamic range. To this end we used a series of different exposure times, ranging from 1 millisecond to 4 seconds, and assembled a high-dynamic-range image using a modified version of Debevec and Malik's technique [4]. To reduce the effects of stray light and fixed-pattern CCD noise, we subtracted a dark image, taken with the illumination beam blocked just before the focusing lens (point A in Figure 5(a)), from each measurement and reference image. The resulting images had a dynamic range of around 10^5 (the small amount of total energy in the image reduces the effects of lens and camera flare, allowing higher dynamic range than might otherwise be possible).

To interpret the measurements, we examined only a 1D slice of each measurement image, corresponding to a line on the surface through the illumination point and perpendicular to the camera's view direction. Under the assumption that light exits diffusely[1], the pixel values p_i in this slice (see Figure 6 for an example) are measurements of radiant exitance as a function of distance on the surface. Since R_d gives the ratio of this quantity to Φ, $p_i = K\Phi R_d(r_i)$, where K is an unknown constant. To eliminate the scale factor, we also took a reference image with the sample replaced by a white ideal diffuse reflector (Labsphere Spectralon, reflectance > 0.99). By summing all the pixels in this image, we can integrate the radiant exitance to get the total flux exiting the surface, which for this special material is equal to the incident flux Φ. With the same constant K as above, this sum is $K\Phi/A$, where A is the (known) area on the sample's surface subtended by one pixel. The measured value for $R_d(r_i)$ can then be computed as $p_i/(K\Phi)$.

In principle, σ_a and σ_s' can be determined by fitting the relative reflectance curve with Equation 4 over a range of distances far enough from the illumination point to allow the use of diffusion theory [8]. However, we found this fitting problem to be ill-conditioned enough that the uncertainty in the resulting parameters led to too much uncertainty in the appearance of the material, especially the total diffuse reflectance.

We remove this ill-conditioning by measuring the total diffuse reflectance R (which is the sum of the measurement image divided by the sum of the reference image) and computing the least-squares fit subject to the constraint $\int R_d \, dA = R$.

Figure 6 shows how these measurements confirm the diffusion theory for a sample of white marble (only the camera's green channel is shown). Fitting the theory (solid line) to the data (points) led to the parameters $\sigma_a = 0.0041$/mm, $\sigma_s' = 2.6$/mm. The reflectance computed by a Monte Carlo simulation using these values (dashed line) confirms the correctness of the computed parameters. Fitted values for several other materials appear in the table in Figure 5(b). Note, that we used empirical values for the index of refraction for most of the materials. Also note that the diffusion theory is assuming that $\sigma_s \gg \sigma_a$, and as such the parameters for the relatively opaque materials (such as the blue wavelength in ketchup) may be less accurate.

[1]We verified this assumption for marble by examining the reflectance for different outgoing angles, and it closely resembled a Lambertian material scaled by a Fresnel transmission term.

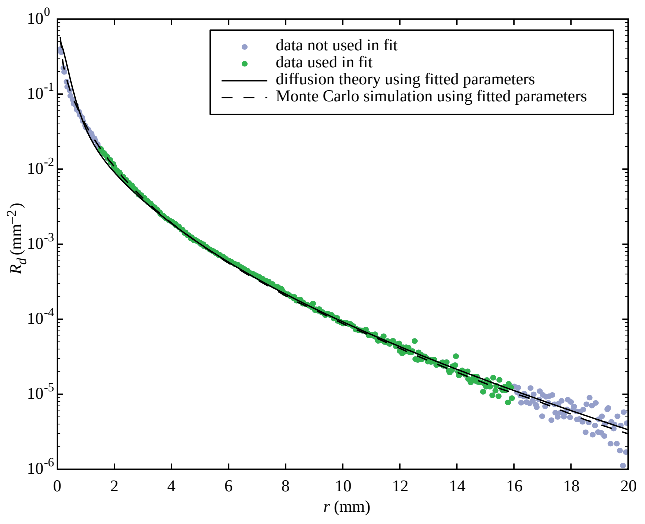


Figure 6: Measurements for marble (green wavelength band) plotted with fit to diffusion theory and confirming Monte Carlo simulation.

4 Rendering Using the BSSRDF

The BSSRDF model derived in the theory section only applies to semi-infinite homogeneous media. A similar derivation is not possible in the presence of arbitrary geometry and texture variation. However, we can use some of the intuition behind the theory to extend it to a practical model for computer graphics. Specifically, we need to consider:

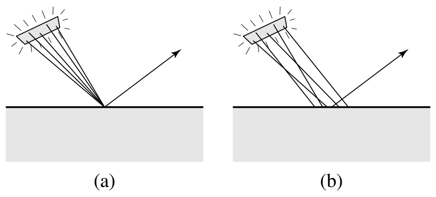


Figure 7: (a) Sampling a BRDF (traditional sampling), (b) sampling a BSSRDF (the sample points are distributed both over the surface as well as the light).

- Efficient integration of the BSSRDF including importance sampling
- Single scattering evaluation for arbitrary geometry
- Diffusion approximation for arbitrary geometry
- Texture (spatial variation on the object surface).

In this section we explain how to do this in a ray-tracing context.

Integrating the BSSRDF: At each ray-object intersection traditional lighting models (based on BRDFs) need just a point and a normal to compute the outgoing radiance (Figure 7(a)). For the BSSRDF it is necessary to integrate the incoming lighting over an area of the surface (Figure 7(b)). We do this by stochastically sampling the location of *both* endpoints of the shadow ray — this can be seen as an extension of the classical distribution ray tracing technique for sampling area light sources [2]. To efficiently sample locations on the surface we exploit the exponential falloff in the diffusion term and the single scattering term. We sample the two terms of the BSSRDF separately, since the single scattering sample locations must be along the refracted outgoing ray whereas the diffusion samples should be distributed around x_o.

More specifically, for the diffusion term, we use standard Monte Carlo techniques to randomly sample the surface with density ($\sigma_{tr} e^{-\sigma_{tr} d}$) at some distance d from x_o.

Single scattering is reparameterized since the incoming ray and the outgoing ray must intersect. Our technique is explained in the following section.

Single scattering evaluation for arbitrary geometry: Single scattering is evaluated using Monte Carlo integration along the refracted outgoing ray. We pick a random distance, $s'_o = \log(\xi)/\sigma_t(x_o)$, along the refracted outgoing ray. Here $\xi \in]0,1]$ is a uniformly distributed random number. For this sample location we compute the outscattered radiance as:

$$L_o^{(1)}(x_o,\vec{\omega}_o)=\frac{\sigma_s(x_o)Fp(\vec{\omega}_i\cdot\vec{\omega}_o)}{\sigma_{tc}}e^{-s'_i\sigma_t(x_i)}e^{-s'_o\sigma_t(x_o)}L_i(x_i,\vec{\omega}_i).$$

Here s'_i is the distance that the sample ray moves through the material. Optimizing this equation to sample direct illumination (with shadow rays) is difficult for arbitrary geometry since it requires finding the point at the surface where the shadow ray is refracted. However, in practice a good approximation can be found by using a shadow ray that does not refract at the surface — this assumes that the light source is far away compared to the mean free path of the medium. We can use Snell's law to estimate the true refracted distance through the medium of the incoming ray:

$$s'_i = s_i \frac{|\vec{\omega}_i\cdot\vec{n}_i|}{\sqrt{1-\left(\frac{1}{\eta}\right)^2(1-|\vec{\omega}_i\cdot\vec{n}(x_i)|^2)}} \; .$$

Here s_i is the observed distance and s'_i is the refracted distance.

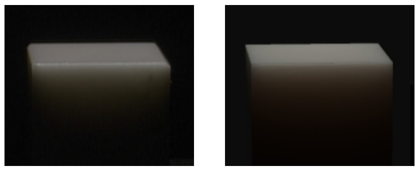


Figure 8: Scattering of laser light in a marble block. The marble block is 40mm. wide and has a significant amount of subsurface scattering. The picture on the left is a photograph of the marble block, and the picture on the right is a synthetic rendering of a similarly sized cube using the BSSRDF model and the measured scattering properties of the marble. Note how the appearance of the two images is very similar.

Diffusion approximation for arbitrary geometry: An important component of the diffusion approximation is the use of the dipole source. If the geometry is locally flat we can get a very good approximation by using a similar dipole source configuration as that for flat materials (i.e., we always place the light source $1/\sigma'_t$ straight below x_i). Special care must be taken in the presence of highly curved surfaces; we handle this case by always evaluating the diffusion term with a minimum distance of $1/\sigma'_t$. In this way we eliminate singularities at sharp edges where the source can be placed arbitrarily close to x_o. We found this approach to work very well in our experiments.

Texture: We approximate textured materials by making a few small changes to the usage of the BSSRDF. We only consider texture variation at the surface — effects due to volumetric texture variation would require a full participating media simulation. For the diffusion approximation we always use the material parameters at x_i, which ensures a natural local blending of the texture properties. For the single scattering term we use $\sigma_s(x_o)$ and $\sigma_t(x_o)$ along the refracted outgoing ray, and $\sigma_t(x_i)$ along the refracted incident ray. This variation is included in Equation 6.

5 Results

We have implemented the BSSRDF model in a Monte Carlo ray tracer, and in this section we will present a number of experimental results obtained with this implementation. All simulations have been done on a dual 800MHz Pentium III PC running Linux and the images have been rendered with 4 samples per pixel and a width of 1024 pixels.

Our first simulation is shown in Figure 8, which compares a side photograph of a marble cube illuminated from above with a synthetic rendering. The synthetic image is rendered using the BSSRDF model and the measured parameters for marble (from the table in Figure 5). We only used a simple cube to approximate the rounded marble block, so there are natural visible differences along the edges. Nonetheless, the BSSRDF model faithfully renders the appearance including the scattered light exiting from the side of the marble cube.

Figure 9 shows several different simulations of subsurface scattering in a marble bust (1.3 million triangles) illuminated from behind. The BSSRDF simulation mostly matches the appearance of the full Monte Carlo simulation, yet is significantly faster (5 minutes vs. 1250 minutes). The hair at the back of the head is slightly darker in the BSSRDF simulation; we believe this is due to the forced $1/\sigma'_t$ distance in the diffusion approximation. A similar rendering was done using photon mapping in [5] in roughly 12 minutes (scaled to the speed of our computer). However, the photon mapping method requires a full 3D-description of the material, it requires memory to store the photons, and it becomes costly for

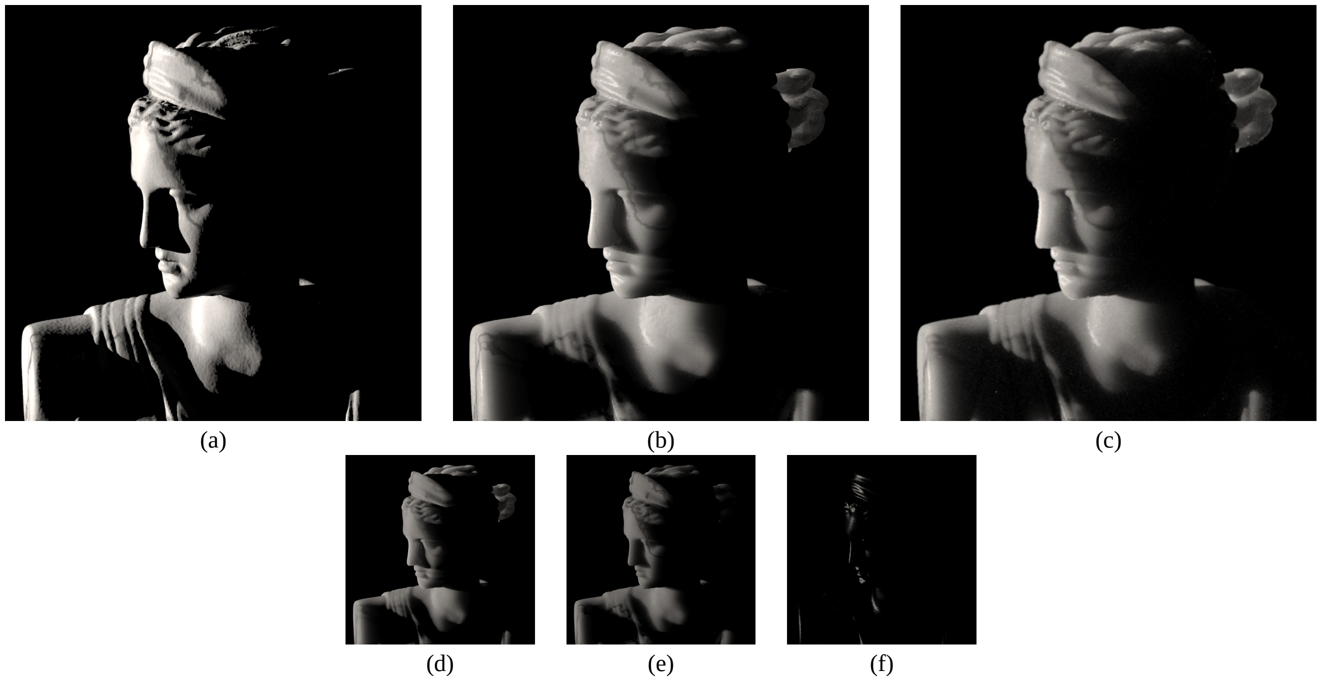


Figure 9: A simulation of subsurface scattering in a marble bust. The marble bust is illuminated from behind and rendered using: (a) the BRDF approximation (in 2 minutes), (b) the BSSRDF approximation (in 5 minutes), and (c) a full Monte Carlo simulation (in 1250 minutes). Notice how the BSSRDF model matches the appearance of the Monte Carlo simulation, yet is significantly faster. The images in (d–f) show the different components of the BSSRDF: (d) single scattering term, (e) diffusion term, and (f) Fresnel term.

highly scattering materials (such as milk and skin).

A particularly interesting aspect of the BSSRDF simulation is that it is able to capture the smooth appearance of the marble surface. In comparison the BRDF simulation gives a very hard appearance where even tiny bumps on the surface are visible (this is a classic problem in realistic image synthesis where objects often look hard and unreal).

For the marble we used synthetic scattering and absorption coefficients, since we wanted to test the difficult case when the average scattering albedo is 0.5 (here the contribution from diffusion and single scattering is approximately the same). Figure 9 demonstrates how the sum of both single scattering and the diffusion term is necessary to match the Monte Carlo simulation.

Figure 10 contains three renderings of milk. The first rendering uses a diffuse reflection model; the others use the BSSRDF model and our measurements for skim milk and whole milk. Notice how the diffuse milk looks unreal and too opaque compared to the BSSRDF images, even though multiple scattering dominates and the radiant exitance due to subsurface scattering is very diffuse. It is interesting that the BSSRDF simulations are capable of capturing the subtle details in the appearance of milk, making the milk look more bluish at the front and more reddish at the back. This is due to Rayleigh scattering that causes shorter wavelengths of light to be scattered more than longer wavelengths.

Skin is a material that is particularly difficult to render using methods that simulate subsurface scattering by sampling ray paths through the material. This is due to the fact that skin is highly scattering (typical albedo is 0.95) and also very anisotropic (typical average cosine of the scattering angle is 0.85). Both of these properties mean that the average number of scattering events of a photon is very high (often more than 100). In addition skin is very translucent, and it cannot be rendered correctly using a BRDF (see Figure 11). A complete skin model requires multiple layers, but a reasonable approximation can be obtained using just one layer. In Figure 11 we have rendered a simple face model using the BSSRDF and our measured values for skin (skin1). Here we also used the Henyey-Greenstein phase function [11] with $g = 0.85$ as the estimated mean cosine of the scattering angle. The skin measurements are from an arm (which is likely more translucent than skin on the face), but the overall appearance is still realistic considering the lack of spatial variation (texture). The BSSRDF gives the skin a soft appearance, and it renders the color bleeding in the shadow region below the nose. Here, the absorption by blood is particularly noticeable as the light that scatters deep in the skin is redder. For this simulation the diffusion term is much larger than the single scattering term. This means that skin reflects light fairly diffusely, but also that internal color bleeding is an important factor. The BRDF image was rendered in 7 minutes, the BSSRDF image was rendered in 17 minutes.

6 Conclusion and Future Work

In this paper we have presented a new practical BSSRDF model for computer graphics. The model combines a dipole diffusion approximation with an accurate single scattering computation. We have shown how the model can be used to measure the scattering properties of translucent materials, and how the measured values can be used to reproduce the results of the measurements as well as synthetic renderings. We evaluate the BSSRDF by sampling the incoming light over the surface, and we demonstrate how this technique is capable of capturing the soft and smooth appearance of translucent materials.

In the future we plan to extend the model to multiple layers as well as include support for efficient global illumination.

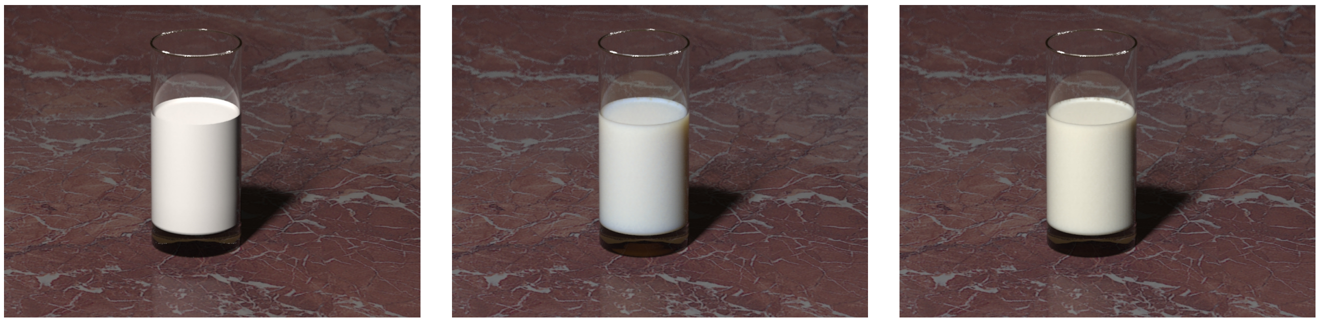


Figure 10: A glass of milk: (a) diffuse (BRDF), (b) skim (BSSRDF) and (c) whole (BSSRDF). (b) and (c) are using our measured values. The rendering times are 2 minutes for (a), and 4 minutes for (b) and (c); this includes caustics and global illumination on the marble table and a depth-of-field simulation.

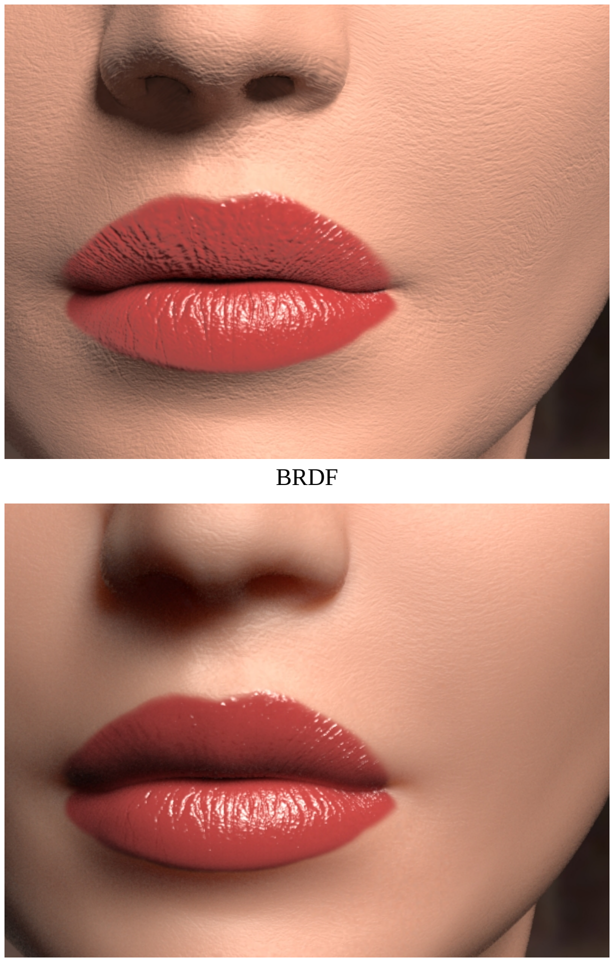


Figure 11: A face rendered using the BRDF model (top) and the BSSRDF model (bottom). We used our measured values for skin (skin1) and the same lighting conditions in both images (the BRDF image also includes global illumination). The face geometry has been modeled by hand; the lip-bumpmap is handpainted, and the bumpmap on the skin is based on a gray-scale macro photograph of a piece of skin. Even with global illumination the BRDF gives a hard appearance. Compare this to the faithful soft appearance of the skin in the BSSRDF simulation. In addition the BSSRDF captures the internal color bleeding in the shadow region under the nose.

7 Acknowledgements

Special thanks to Steven Stahlberg for modeling the face. Thanks to the SIGGRAPH reviewers and to Maryann Simmons and Heidi Marschner for helpful comments on the manuscript. This research was funded in part by the National Science Foundation Information Technology Research grant (IIS-0085864). The first author was also supported by DARPA (DABT63-95-C-0085), and the second author was also supported by Honda North America, Inc.

References

[1] S. Chandrasekhar. *Radiative Transfer*. Oxford Univ. Press, 1960.

[2] R. L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. In *ACM Computer Graphics (SIGGRAPH'84)*, volume 18, pages 137–145, July 1984.

[3] P. Debevec, T. Hawkins, C. Tchou, H. Duiker, W. Sarokin, and M. Sagar. Acquiring the reflectance field of a human face. In *Computer Graphics Proceedings, Annual Conference Series, 2000*, pages 145–156, July 2000.

[4] P. E. Debevec and J. Malik. Recovering high dynamic range radiance maps from photographs. In *Computer Graphics Proceedings, Annual Conference Series, 1997*, pages 369–378, August 1997.

[5] J. Dorsey, A. Edelman, H. W. Jensen, J. Legakis, and H. K. Pedersen. Modeling and rendering of weathered stone. In *Computer Graphics Proceedings, Annual Conference Series, 1999*, pages 225–234, August 1999.

[6] G. Eason, A. Veitch, R. Nisbet, and F. Turnbull. The theory of the backscattering of light by blood. *J. Physics*, 11:1463–1479, 1978.

[7] W. G. Egan and T. W. Hilgeman. *Optical Properties of Inhomogeneous Materials*. Academic Press, New York, 1979.

[8] T. J. Farell, M. S. Patterson, and B. Wilson. A diffusion theory model of spatially resolved, steady-state diffuse reflectance for the noninvasive determination of tissue optical properties in vivo. *Med. Phys.*, 19:879–888, 1992.

[9] R. A. Groenhuis, H. A. Ferwerda, and J. J. Ten Bosch. Scattering and absorption of turbid materials determined from reflection measurements. 1: Theory. *Applied Optics*, 22:2456–2462, 1983.

[10] P. Hanrahan and W. Krueger. Reflection from layered surfaces due to subsurface scattering. In *ACM Computer Graphics (SIGGRAPH'93)*, pages 165–174, August 1993.

[11] L. G. Henyey and J. L. Greenstein. Diffuse radiation in the galaxy. *Astrophysics Journal*, 93:70–83, 1941.

[12] A. Ishimaru. *Wave Propagation and Scattering in Random Media*, volume 1. Academic Press, New York, 1978.

[13] G. Kortum. *Reflectance Spectroscopy*. Springer-Verlag, 1969.

[14] F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, and T. Limperis. Geometric considerations and nomenclature for reflectance. Monograph 161, National Bureau of Standards (US), October 1977.

[15] M. Pharr and P. Hanrahan. Monte Carlo evaluation of non-linear scattering equations for subsurface reflection. In *Computer Graphics Proceedings, Annual Conference Series, 2000*, pages 75–84, July 2000.

[16] L. Reynolds, C. Johnson, and A. Ishimaru. *Applied Optics*, 15:2059, 1976.

[17] J. Stam. Multiple scattering as a diffusion process. In *Eurographics Rendering Workshop 1995*. Eurographics, June 1995.

This reprint originally appeared as:

Henrik Wann Jensen and Juan Buhler. A rapid hierarchical rendering technique for translucent materials. In *Proceeding of ACM SIGGRAPH 2002*, pages 576-581, July 2002.

A Rapid Hierarchical Rendering Technique for Translucent Materials

Henrik Wann Jensen
Stanford University

Juan Buhler
PDI/DreamWorks

Abstract

This paper introduces an efficient two-pass rendering technique for translucent materials. We decouple the computation of irradiance at the surface from the evaluation of scattering inside the material. This is done by splitting the evaluation into two passes, where the first pass consists of computing the irradiance at selected points on the surface. The second pass uses a rapid hierarchical integration technique to evaluate a diffusion approximation based on the irradiance samples. This approach is substantially faster than previous methods for rendering translucent materials, and it has the advantage that it integrates seamlessly with both scanline rendering and global illumination methods. We show several images and animations from our implementation that demonstrate that the approach is both fast and robust, making it suitable for rendering translucent materials in production.

Keywords: Subsurface scattering, BSSRDF, reflection models, light transport, diffusion theory, global illumination, realistic image synthesis

1 Introduction

Translucent materials are frequently encountered in the natural world. Examples include snow, plants, milk, cheese, meat, human skin, cloth, marble, and jade. The degree of translucency may vary, but the characteristic appearance is distinctly smooth and soft as a result of light scattering inside the objects, a process known as subsurface scattering. Subsurface scattering diffuses the scattered light and blurs the effect of small geometric details on the surface, softening the overall look. In addition, scattered light can pass through translucent objects; this is particularly noticeable when the objects are lit from behind. To render these phenomena and capture the true appearance of translucent materials it is therefore necessary to simulate subsurface scattering.

Traditionally subsurface scattering has been approximated as Lambertian diffuse reflection. This was later improved by Hanrahan and Krueger [1993] with an analytic term for single scattering in order to account for important directional effects. They also proposed a method for simulating subsurface scattering by tracing photons through the material, but in the end they used a BRDF (Bidirectional Reflectance Distribution Function [Nicodemus et al. 1977]) to represent the final model. A BRDF only accounts for scattering at a single point, and it cannot be used to simulate light transport within the material between different points on the surface. This requires treating the material as a participating medium with a surface. This was done by Dorsey et al. [1999] who used photon mapping to simulate subsurface scattering in weathered stone. Pharr and Hanrahan [2000] introduced the concept of scattering equations and demonstrated how this concept could be used to simulate subsurface scattering more efficiently than traditional Monte Carlo ray tracing.

More recently, Koenderink and van Doorn [2001] and Jensen et al. [2001] proposed modeling the scattering of light in translucent materials as a diffusion process. The diffusion approximation works particularly well in highly scattering media where traditional Monte Carlo ray tracing becomes very expensive [Stam 1995]. Jensen et al. [2001] suggested a simple analytical dipole diffusion approximation and found this model to be in good agreement with measurements of light scattered from translucent materials. They used this approximation to formulate a complete BSSRDF (Bidirectional Scattering Surface Reflectance Distribution Function [Nicodemus et al. 1977]), which relates outgoing radiance at a point to incident flux at all points on the surface. Finally, they evaluate the BSSRDF by sampling the incident flux on the surface.

The BSSRDF approximation [Jensen et al. 2001] is much faster than Monte Carlo photon tracing. However, since it requires sampling the incident flux distribution at the surface, it is still more expensive to evaluate than a traditional BRDF. It is particularly expensive for highly translucent materials where light can scatter a long distance within the material. Another difficulty with the approach is that it only includes internal scattering in the material due to direct illumination from the light sources. It is not obvious how to extend the sampling technique to include global illumination as well.

In this paper we introduce a fast and general two-pass rendering technique for translucent materials. Our approach is based on two key ideas. The first idea is to decouple of the computation of the incident illumination from the evaluation of the BSSRDF by using a two-pass approach. In the first pass, we compute the irradiance at selected points on the surface, and in the second pass we evaluate a diffusion approximation using the pre-computed irradiance samples. The second idea is to use a rapid hierarchical evaluation of the diffusion approximation using the pre-computed irradiance samples. This approach is substantially faster than directly sampling the BSSRDF since it only evaluates the incident illumination once at a given surface location, and it is particularly efficient for highly translucent materials where sampling the BSSRDF is costly. To evaluate the irradiance, we can use standard rendering techniques including scanline rendering and global illumination methods. This means that we can compute the effects of indirect illumination on translucent materials. Furthermore, our results do not suffer from any high-frequency Monte Carlo sampling noise since the hierarchical evaluation is deterministic. This is a great advantage for animations where this type of noise is particularly noticeable.

Another contribution of this paper is a reformulation of the scattering parameters for translucent materials. We show how the intrinsic scattering properties of translucent materials can be computed from two intuitive parameters: a diffuse reflectance and an average scattering distance. Finally, we show several results from our implementation of the method in a scanline renderer as well as a Monte Carlo ray tracer. Our results indicate that the hierarchical evaluation technique is fast and robust, and capable of rendering images and animations of translucent objects in complex lighting environments.

2 Light Diffusion in Translucent Materials

The scattering of light within a medium is described by the radiative transport equation [Chandrasekhar 1960]:

$$(\vec{\omega} \cdot \nabla) L(x, \vec{\omega}) = -\sigma_t L(x, \vec{\omega}) + \sigma_s L_i(x, \vec{\omega}) + s(x, \vec{\omega}). \quad (1)$$

Here, L is the radiance, s is a source term, σ_s is the scattering coefficient, σ_a is the absorption coefficient, σ_t is defined as $\sigma_a + \sigma_s$, and L_i is the in-scattered radiance:

$$L_i(x, \vec{\omega}) = \int_{4\pi} p(\vec{\omega}, \vec{\omega}') L(x, \vec{\omega}') d\vec{\omega}'. \quad (2)$$

The phase function, p, specifies the spherical distribution of the scattered light. It is normalized, $\int_{4\pi} p(\vec{\omega}, \vec{\omega}') d\vec{\omega}' = 1$, and we assume it only depends on the cosine of the scattering angle, $p(\vec{\omega}, \vec{\omega}') = p(\vec{\omega} \cdot \vec{\omega}')$. The mean cosine, g, of the scattering angle is:

$$g = \int_{4\pi} p(\vec{\omega}, \vec{\omega}')(\vec{\omega} \cdot \vec{\omega}') d\vec{\omega}'. \quad (3)$$

The value of $g \in [-1, 1]$ indicates the type of scattering in the medium. $g = 0$ is isotropic scattering, $g < 0$ is backwards scattering and $g > 0$ is forward scattering. Most translucent materials are strongly forward scattering with $g > 0.7$ (skin for example has $0.7 < g < 0.9$ [Gemert et al. 1989]). Such strongly peaked phase functions are costly to simulate in media with multiple scattering since the probability of sampling in the direction of the light sources will be low in most situations. The difficulty of sampling further increases with the distance to the light sources. In this case we can benefit from a powerful technique known as the *similarity of moments* [Wyman et al. 1980], which allows us to change the scattering properties of the medium without significantly influencing the actual distribution of light. Specifically, we can modify the medium to have isotropic scattering ($g = 0$) by changing the scattering coefficient to

$$\sigma_s' = (1 - g)\sigma_s, \quad (4)$$

where σ_s' is the *reduced* scattering coefficient. The absorption coefficient remains unchanged, and we get the reduced extinction coefficient $\sigma_t' = \sigma_s' + \sigma_a$.

Equation 1 is a five-dimensional integro-differential equation, and even in media with isotropic scattering it is in most cases difficult to solve. One approach is to expand radiance into a truncated series of spherical harmonics. For this purpose we divide the radiance into two components: the unscattered radiance, L_u, and the scattered (diffuse) radiance, L_d. The unscattered radiance is reduced as a function of the distance traveled through the medium [Ishimaru 1978]:

$$L_u(x + \Delta x, \vec{\omega}) = e^{-\sigma'_t \Delta x} L_u(x, \vec{\omega}). \quad (5)$$

The average distance at which the light is scattered, the mean-free path, is $\ell_u = 1/\sigma_t'$.

The diffusion approximation uses the first four terms of the spherical harmonic expansion to represent L_d:

$$L_d(x, \vec{\omega}) \approx F_t(x) + \frac{3}{4}\pi \vec{E}(x) \cdot \vec{\omega}. \quad (6)$$

The 0th-order spherical harmonic, the radiant fluence, F_t, is $F_t(x) = \int_{4\pi} L_d(x, \vec{\omega}') d\vec{\omega}'$, and the 3 terms of the 1st-order spherical harmonic, the vector irradiance, $\vec{E}$, is $\vec{E} = \int_{4\pi} L_d(x, \vec{\omega}') \vec{\omega}' d\vec{\omega}'$. Note that L_d cannot be purely diffuse as this would result in zero flux within the medium. Instead L_d is approximated as being mostly diffuse, but with a preferential direction (as indicated by $\vec{E}$) to the overall flow of the flux.

The diffusion approximation is particularly effective in highly scattering media at some distance from the light sources as well as in regions with rapidly changing scattering properties. This is due to the natural smoothing resulting from multiple scattering [Stam 1995]. More precisely, the diffusion approximation has been shown [Furutso 1980] to be accurate when $\sigma_a/\sigma_t \ll 1 - g^2$.

Applying the diffusion approximation (Equation 6) to the radiative transport equation (Equation 1) yields the diffusion equation (the details of the derivation can be found in [Ishimaru 1978]):

$$\frac{1}{3\sigma_t'} \nabla^2 F_t(x) = \sigma_a F_t(x) - S_0(x) + \frac{1}{\sigma_t'} \nabla \cdot \vec{S}_1(x). \quad (7)$$

Here, S_0 and S_1 represents the 0th- and the 1st-order spherical harmonics expansion of the source term, similar to the expansion for diffuse radiance.

The diffusion equation can be solved analytically for special cases [Ishimaru 1978], or by using a multigrid approach [Stam 1995]. In the case of translucent materials, we are interested in the outgoing radiance at the material surface as a function of the incoming radiance. Jensen et al. [2001] use a dipole diffusion approximation for a point source in a semi-infinite medium. The point source is an approximation of an incoming beam of light for which it is assumed that all light scatters at a depth of one mean-free path below the surface. The dipole diffusion approximation results in the following expression for the radiant exitance, M_o, at surface location x_o due to incident flux, $\Phi_i(x_i)$, at x_i:

$$dM_o(x_o) = d\Phi_i(x_i) \frac{\alpha'}{4\pi} \left\{ C_1 \frac{e^{-\sigma_{tr} d_r}}{d_r^2} + C_2 \frac{e^{-\sigma_{tr} d_v}}{d_v^2} \right\}, \quad (8)$$

where

$$C_1 = z_r \left(\sigma_{tr} + \frac{1}{d_r} \right) \text{ and } C_2 = z_v \left(\sigma_{tr} + \frac{1}{d_v} \right). \quad (9)$$

Here, $\alpha' = \sigma_s'/\sigma_t'$ is the reduced albedo, $\sigma_{tr} = \sqrt{3\sigma_a \sigma_t'}$ is the effective transport extinction coefficient, $d_r = \sqrt{r^2 + z_r^2}$ is the distance to the real light source, $d_v = \sqrt{r^2 + z_v^2}$ is the distance to the virtual source, $r = ||x_o - x_i||$ is the distance from x_o to the point of illumination, and $z_r = \ell_u$ and $z_v = \ell_u(1 + 4/3A)$ are the distance from the the dipole lights to the surface (shown in Figure 2). Finally, the boundary condition for mismatched interfaces is taken into account by the A term which is computed as $A = (1 + F_{dr})/(1 - F_{dr})$, where the diffuse Fresnel term, F_{dr} is approximated from the relative index of refraction η by [Groenhuis et al. 1983]:

$$F_{dr} = -\frac{1.440}{\eta^2} + \frac{0.710}{\eta} + 0.668 + 0.0636\eta. \quad (10)$$

In addition to Equation 8 the BSSRDF includes a single scattering term (see [Jensen et al. 2001] for the details).

2.1 The Importance of Multiple Scattering

The diffuse term is the most costly to sample for translucent materials since it depends on lighting from a large fraction of the material surface. We can approximate the average distance, $\ell_d = 1/\sigma_{tr}$,

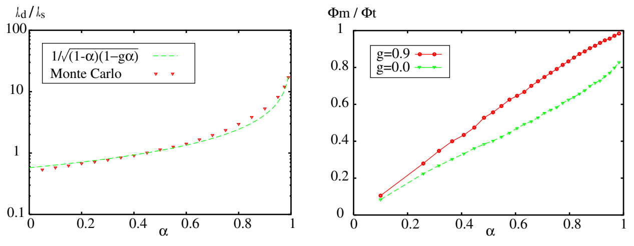


Figure 1: These graphs show the effect of increasing the scattering albedo of the material. The left graph shows the average scattering distance for diffuse radiance divided by the mean-free path for single scattering (for $g = 0.9$) as predicted by Equation 11 and estimated using a Monte Carlo photon simulation. The graph on the right shows the fraction of the radiant exitance that is due to multiple scattering (estimated with a Monte Carlo photon simulation). The important thing to notice in the two graphs is that the diffuse radiance scatters much further, and that it becomes increasingly important as the albedo gets closer to one.

along the surface that the diffused radiance scatters by assuming that the exponential term dominates in Equation 8. By dividing this distance with the mean-free path, $\ell_s = 1/\sigma_t$, of single-scattered light, we can estimate the relative scattered distance of the two within the medium:

$$\frac{\ell_d}{\ell_s} = \frac{\sigma_t}{\sigma_{tr}} = \frac{1}{\sqrt{3(1-\alpha)(1-g\alpha)}}. \tag{11}$$

Note how the ratio depends only on the albedo, α, and the scattering anisotropy, g. Figure 1 shows a plot of this equation and a comparison with a Monte Carlo photon simulation. For the photon simulation, we traced photons from random directions towards a translucent material and recorded the average distance at which the photons left the surface again after scattering inside the material. This distance divided by ℓ_s is shown in the graph. For the simulation we used the Henyey-Greenstein phase function [Henyey and Greenstein 1941] and the photons are scattered using the approach described by Hanrahan and Krueger [1993]. Despite several assumptions about the average scattering distance, it can be seen that the predictions of Equation 11 are surprisingly accurate. For both simulations the ratio rapidly increases as the albedo approaches one as a consequence of the increasing number of scattering events. From the measurements in [Jensen et al. 2001] we can see that all of the materials have an albedo close to one. As an example, red wavelengths in skim milk (assuming $g = 0.9$) have a scattering albedo of $\alpha \approx 0.9998$, which gives a ratio $\ell_d/\ell_s \approx 129$. This means that the average distance traveled by diffuse radiance is 129 times larger than the average distance traveled by unscattered radiance. In effect this means that single scattering is substantially more localized than diffuse scattering.

The importance of multiple scattering increases with the albedo of the material. To further investigate how important multiple scattered light is for translucent materials, we performed another Monte Carlo photon simulation. In this simulation we traced photons from random directions towards the surface scattering medium. At the surface we recorded the radiant exitance from the photons that scattered in the medium. We used an index of refraction of 1.3 for the medium (the results are very similar for other values). Two important parameters for the medium are the scattering anisotropy and the scattering albedo. The right graph in Figure 1 shows the fraction of the radiant exitance from the material due to multiple scattered light as a function of the albedo. Note, how the fraction gets close to 1.0 for the forward scattering material, and close to 0.9 for a material with isotropic scattering.

3 A Two-Pass Technique for Evaluating the Diffusion Approximation

As shown in the previous section, the radiant exitance from highly scattering translucent materials is dominated by photons that have scattered multiple times inside the material. Jensen et al. [2001] compute the contribution from multiple scattering by sampling the irradiance at the material surface and evaluating the diffusion approximation — in effect convolving the reflectance profile predicted by the diffusion approximation with the incident illumination. Even though the diffusion approximation is a very effective way of approximating multiple scattering, this sampling technique becomes expensive for highly translucent materials. The reason for this is that the sampled surface area grows and needs more samples as the material becomes more translucent.

The key idea for making this process faster is to decouple the computation of irradiance from the evaluation of the diffusion approximation. This makes it possible to reuse irradiance samples for different evaluations of the diffusion equation. For this purpose, we use a two-pass approach in which the first pass consists of computing the irradiance at selected points on the surface, and the second pass is evaluating the diffusion approximation using the precomputed irradiance values. For the second pass we exploit the decreasing importance of distant samples and use a rapid hierarchical integration technique.

Pass 1: Sampling the Irradiance

To obtain the sample locations on the surface of a piece of geometry we use Turk's point repulsion algorithm [Turk 1992], which produces a uniform sampling of points on a polygon mesh. We do not change (retile) our mesh as we only need the point locations. To ensure an accurate evaluation of the diffusion approximation we need enough points to account for several factors including the geometry, the variation in the lighting, the scattering properties of the material, and the integration technique. We use the mean-free path, ℓ_u, as the maximum distance between the points on the surface. The approximate number of points that we use for a given object then becomes $A/(\pi\ell_u^2)$, where A is the surface area of the object. This is a conservative estimate, since anything below the mean-free path will be blurred by multiple scattering. However, the sample density should not be much lower since this will result in low-frequency noise in the reconstruction of the diffusion approximation. Note that our reconstruction does not require a uniform sampling since we weight each sample point by the area associated with the point. It would be possible to use other approaches that sample more densely around discontinuities in the irradiance or the geometry.

With each sample point we store the location, the area associated with the point (in the case of uniform sampling, this is simply the surface area divided by the number of points), and a computed irradiance estimate. Since the irradiance is computed at a surface location we can use standard rendering techniques including methods that account for global illumination (such as photon mapping [Jensen 1996] and irradiance caching [Ward et al. 1988]).

Pass 2: Evaluating the Diffusion Approximation

The diffusion approximation can be evaluated directly (using Equation 8) by summing the contribution from all the irradiance samples. However, this approach is costly since most objects have several thousand irradiance samples. Another, strategy would be to only consider nearby "important" points. This approach would work, but it could potentially leave out important illumination, and for accurate evaluations it would still need hundreds of irradiance samples (e.g. our sampling produces roughly 300 samples within a disc with a radius of 10 mean-free paths). Instead we use a hierarchical evaluation technique which takes into account the contribution from all irradiance samples by clustering distant samples to make this evaluation fast. The exponential shaped fall-off in the diffusion approximation makes the hierarchical approach very efficient. The concept is similar to the hierarchical approaches used for N-body problems [Appel 1985].

Several different hierarchical structures can be used for the irradiance samples. We use an octree in our implementation. Each node in the tree stores a representation of illumination in all its child nodes: the total irradiance, E_v, the total area represented by the points, A_v, and the average location (weighted by the irradiance) of the points, $\vec{P}_v$. To increase efficiency we allow up to 8 irradiance samples in a leaf voxel.

The total radiant exitance flux at a location, x, is computed by traversing the octree from the root. For each voxel we check if it is "small enough" or if it is a leaf node that it can be used directly; otherwise all the child nodes of the voxel are evaluated recursively. If the point x is inside the voxel then we always evaluate the child nodes. For all other voxels we need an error criterion that specifies the desired accuracy of the hierarchical evaluation. One option would be to compute the potential contribution from the voxel and decide based on this estimate if the voxel should be subdivided. Unfortunately, this is not trivial — and simply using the center of the points in the voxel is not a good approximation for nearby large voxels. Instead we use a much simpler criterion that is both fast to evaluate and that works well in practice. We base our criteria for subdividing a voxel on an approximation of the maximum solid angle, $\Delta\omega$, spanned by the points in the voxel:

$$\Delta\omega = \frac{A_v}{||\vec{x} - \vec{P}_v||^2}. \tag{12}$$

To decide if a voxel should be subdivided we simply compare $\Delta\omega$ to a value ϵ which controls the error. If $\Delta\omega$ is larger than ϵ then the children of the voxel are evaluated; otherwise the voxel is used directly. Another option would be to check the solid angle of the voxel itself; however, using the area of the points makes the evaluation faster, since we can use the clustered values for large voxels with just a few irradiance samples (e.g. large voxels that just barely intersect the surface).

The radiant exitance due to a voxel is evaluated using the clustered values for the voxel, or if it is a leaf-voxel by summing the contribution from each of the points in the voxel. The radiant exitance, $M_{o,p}(x)$ at x from a given irradiance sample is computed using the dipole diffusion approximation

$$M_{o,p}(x) = F_{dt}(x)\frac{dM_o(x)}{\alpha' d\Phi_i(\vec{P}_p)}E_p A_p, \tag{13}$$

where P_p is the location of the irradiance sample(s), E_p is the irradiance at that location, and A_p is the area of the location. $dM_o(||x - \vec{P}_p||_2)/(\alpha' d\Phi_i(\vec{P}_p))$ is computed using Equation 8. Notice that we scale each irradiance sample by the diffuse Fresnel transmittance, $F_{dt} = 1 - F_{dr}$ (F_{dr} is computed using Equation 10). This is necessary when approximating the irradiance by the dipole source. We could have scaled the contribution from each of the sample rays used to compute the irradiance by the true Fresnel transmittance, but by using the diffuse (Lambertian) assumption we can benefit from fast rendering techniques for diffuse materials (e.g. caching techniques such as photon maps [Jensen 1996] and irradiance caching [Ward et al. 1988]). The dipole approximation for an irradiance sample is illustrated in Figure 2. Note that this approximation has been derived assuming a semi-infinite medium. In the presence of complex geometry (e.g. curved surfaces or thin geometry) we use the same techniques as Jensen et al. [2001] to ensure numerical stability.

The result of traversing and evaluating the voxels is an estimate of the total (diffuse) radiant exitance, M_o at x, which we convert into radiance, L_o:

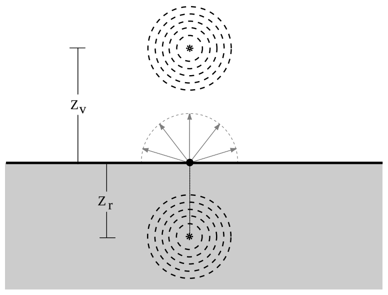


Figure 2: For each point sample we use the dipole diffusion approximation to compute the radiant exitance.

$$L_o(x,\vec{\omega}) = \frac{F_t(x,\vec{\omega})}{F_{dr}(x)}\frac{M_o(x)}{\pi}. \tag{14}$$

We scale the contribution by the Fresnel transmittance, F_t to account for reflection and transmission at the surface. Since, the diffusion approximation already includes a diffuse Fresnel transmittance we divide by F_{dr}. Alternatively, we could omit the Fresnel terms and assume a diffuse radiance.

4 Reparameterizing the BSSRDF

One difficulty in simulating subsurface scattering is that it is difficult to predict the resulting appearance from a given combination of absorption and scattering coefficients (since their effect is highly non-linear). In this section, we will outline a simple technique for reparameterizing the BSSRDF into using intuitive translucency and reflectivity parameters. These parameters are already present in the computations in the form of the diffuse mean free path ℓ_d and the diffuse reflectance of the material, and they are sufficient for computing the scattering and absorption properties of the material.

First, using the diffuse reflection coefficient (see [Jensen et al. 2001]), we solve for the reduced albedo of the material:

$$R_d = \frac{\alpha'}{2}\left(1 + e^{-\frac{4}{3}A\sqrt{3(1-\alpha')}}\right)e^{-\sqrt{3(1-\alpha')}}. \tag{15}$$

This equation is not easily invertible, but it has a simple monotonic shape in the valid region $\alpha' \in [0:1]$, and we use a few iterations of a simple secant root finder to compute α'.

We know the effective transport coefficient, $\sigma_{tr} \approx 1/\ell_d$, and given the reduced albedo we can find the reduced extinction coefficient:

$$\frac{\sigma_{tr}}{\sigma'_t} = \sqrt{3(1-\alpha')} \quad\longrightarrow\quad \sigma'_t = \frac{\sigma_{tr}}{\sqrt{3(1-\alpha')}} \tag{16}$$

Finally, this gives us the absorption and the reduced scattering coefficients: $\sigma'_s = \alpha'\sigma'_t$ and $\sigma_a = \sigma'_t - \sigma'_s$. If the scattering anisotropy, g, is given then the real extinction and scattering coefficients can be computed as well.

5 Results

In this section we present several results from our implementation of the rendering technique. We have used two different systems to implement the model: A Monte Carlo ray tracer with support for global illumination, and a modified a-buffer renderer used in production. Our timings were recorded on a dual 800MHz Pentium 3 for images with a width of 1024 pixels and 4 samples per pixel.

The first example include several renderings of a translucent marble teapot as shown in Figure 3. All of these images were rendered with the Monte Carlo ray tracer. The left column shows a comparison with the BSSRDF sampling technique by Jensen et al. [2001], and our hierarchical technique under the same lighting conditions (for this comparison we use the BRDF approximation for the single scattering term). The important thing to notice is that the two images are practically indistinguishable except for a small amount of sampling noise in the BSSRDF image. This shows that the hierarchical approach is capable of matching the output of the BSSRDF for a translucent material. However, the hierarchical technique took just 7 seconds to render (including 1 second to compute the irradiance values at the samples), while the BSSRDF sampling took 18 minutes — a factor of 154 speedup in this case. The speedup will be even more dramatic for objects that are more translucent. The top image in the right column shows a glossy teapot illuminated by a high dynamic range environment map [Debevec 1999]. To enhance the translucency effect we made the environment black behind the camera. The render time for the image without glossy reflection is 7 seconds (the rendering time including glossy reflection is 40 seconds). The precomputation time for the irradiance samples (sampling the environment map) was roughly 1 minute. This scene would be extremely costly to render using the BSSRDF sampling approach, since it would involve picking a point on the light source and then sampling in the direction of the teapot — a process where the probability of generating good samples is very low. Finally, the lower right image shows the 150,000 sample locations on the teapot that we used for all images.

Our second example in Figure 4 shows the classic Cornell box global illumination scene with a translucent box. This image was rendered using Monte Carlo ray tracing — we used photon mapping [Jensen 1996] to speed up the computation of global illumination. Note the light scattering through the box, and the color bleeding in the scene. In the precomputation of the indirect illumination for the irradiance samples on the translucent box we used a diffuse assumption in order to account for multiple reflections between the box and the walls. However we do account for translucency when evaluating color bleeding on the wall. For scenes where translucency is important for the indirect illumination of the translucent elements (e.g. light bleeding through a leaf onto a white floor which then reflects back on the leaf) a multi-pass approach could be used where the indirect illumination is computed recursively for the translucent materials. The rendering time for the image was 55 seconds, and the precomputation of the irradiance for 20,000 points on the box was 21 seconds.

Our third example in Figure 6 shows the lower half of a face model rendered using a standard skin shader (on the left) and using a skin shader with support for translucency (on the right). This face model was built before the translucency shader was developed. It uses several textures, which we apply by scaling the 250,000 irradiance samples with filtered texture values (the filter support is equal to the area associated with each irradiance sample). This approach made it possible to replace the previous skin shader with a translucent version. The added translucency vastly increases the realism of the model as it softens the appearance of the skin and naturally simulates effects such as the color bleeding around the nose. The rendering time using the standard skin shader was 16 minutes while the translucent skin shader took 20 minutes (including generating the sample points). A significant advantage of our approach is that it works with all the standard lights used in production such as fill lights, rim lights and key lights. This natural integration of the translucency shader in the production framework made it a natural choice for the main character in "Sprout" (a short animation). Translucency helps depict the small size of the character as shown in Figure 5.

6 Conclusion and Future Work

We have presented an efficient two-pass rendering technique for translucent materials. We combine a point sampling of the irradiance on the surface with a fast hierarchical evaluation of a diffusion approximation. Our approach is particularly efficient for highly translucent materials where the BSSRDF sampling [Jensen et al. 2001] becomes costly, and it integrates seamlessly with both scanline rendering and global illumination methods. Our results demonstrate how the technique is both fast and robust making it suitable for rendering translucent materials in production of computer animations.

Future improvements include extending the approach to translucent materials with a visible internal 3D structure. It would also be useful to investigate the accuracy of the dipole diffusion approximation in the presence of complex geometry.

Another interesting path to explore is interactive rendering of translucent materials. This could be done by further simplifying the evaluation technique so that it can be implemented directly on programmable graphics hardware.

7 Acknowledgments

Thanks to the SIGGRAPH reviewers and to Maryann Simmons, Mike Cammarano, Pat Hanrahan, and Marc Levoy for helpful comments. The first author was supported by NSF ITR (IIS-0085864). The second author was supported by PDI/DreamWorks and by the PDI R&D group and thanks them for supporting the development and publication of these techniques.

References

APPEL, A. 1985. An efficient program for many-body simulations. *SIAM Journal of Scientific Statistical Computing 6*, 85–103.

CHANDRASEKHAR, S. 1960. *Radiative Transfer*. Oxford Univ. Press.

DEBEVEC, P., 1999. St. Peter's Basilica (www.debevec.org/probes/).

DORSEY, J., EDELMAN, A., JENSEN, H. W., LEGAKIS, J., AND PEDERSEN, H. K. 1999. Modeling and rendering of weathered stone. In *Proceedings of SIGGRAPH'99*, 225–234.

FURUTSO, K. 1980. Diffusion equation derived from space-time transport equation. *J. Opt. Soc. Am 70*, 360.

GEMERT, M., JACQUES, S., STERENBORG, H., AND STAR, W. 1989. Skin optics. *IEEE Trans. on Biomedical Eng. 16*, 1146–1156.

GROENHUIS, R. A., FERWERDA, H. A., AND BOSCH, J. J. T. 1983. Scattering and absorption of turbid materials determined from reflection measurements. 1: Theory. *Applied Optics 22*, 2456–2462.

HANRAHAN, P., AND KRUEGER, W. 1993. Reflection from layered surfaces due to subsurface scattering. In *Computer Graphics (SIGGRAPH'93 Proceedings)*, 165–174.

HENYEY, L., AND GREENSTEIN, J. 1941. Diffuse radiation in the galaxy. *Astrophysics Journal 93*, 70–83.

ISHIMARU, A. 1978. *Wave Propagation and Scattering in Random Media*, vol. 1. Academic Press, New York.

JENSEN, H. W., MARSCHNER, S. R., LEVOY, M., AND HANRAHAN, P. 2001. A practical model for subsurface light transport. In *Proceedings of SIGGRAPH 2001*, 511–518.

JENSEN, H. W. 1996. Global illumination using photon maps. In *Rendering Techniques '96*, Springer Wien, X. Pueyo and P. Schröder, Eds., 21–30.

KOENDERINK, J., AND VAN DOORN, A. 2001. Shading in the case of translucent objects. In *Proceedings of SPIE*, vol. 4299, 312–320.

NICODEMUS, F. E., RICHMOND, J. C., HSIA, J. J., GINSBERG, I. W., AND LIMPERIS, T. 1977. Geometric considerations and nomenclature for reflectance. Monograph 161, National Bureau of Standards (US), Oct.

PHARR, M., AND HANRAHAN, P. 2000. Monte carlo evaluation of non-linear scattering equations for subsurface reflection. In *Proceedings of SIGGRAPH 2000*, 75–84.

STAM, J. 1995. Multiple scattering as a diffusion process. In *Eurographics Rendering Workshop 1995*, Eurographics.

TURK, G. 1992. Re-tiling polygonal surfaces. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, vol. 26, 55–64.

WARD, G. J., RUBINSTEIN, F. M., AND CLEAR, R. D. 1988. A ray tracing solution for diffuse interreflection. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, vol. 22, 85–92.

WYMAN, D. R., PATTERSON, M. S., AND WILSON, B. C. 1980. Similarity relations for anisotropic scattering in monte carlo simulations of deeply penetrating neutral particles. *J. Comp. Physics 81*, 137–150.

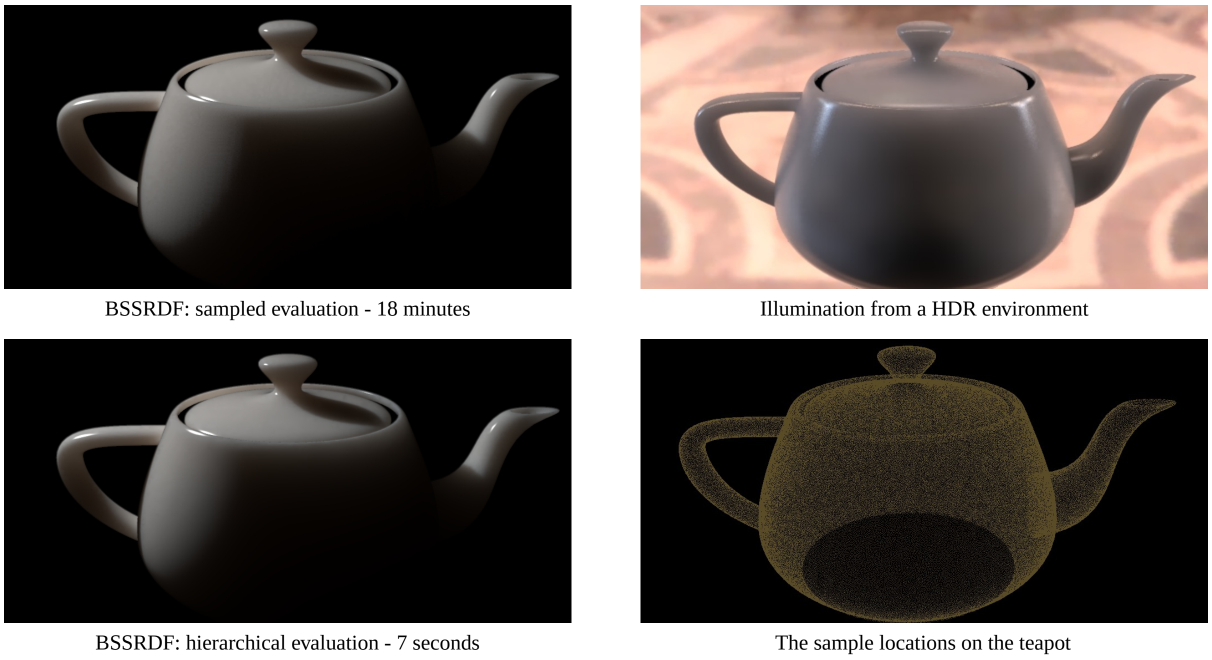

BSSRDF: sampled evaluation - 18 minutes
Illumination from a HDR environment
BSSRDF: hierarchical evaluation - 7 seconds
The sample locations on the teapot

Figure 3: A translucent teapot. On the left we compare our hierarchical BSSRDF evaluation (bottom) to a sampled BSSRDF (top). The top right image shows the teapot in a HDR environment, and the bottom right shows the 150,000 sample points on the teapot.

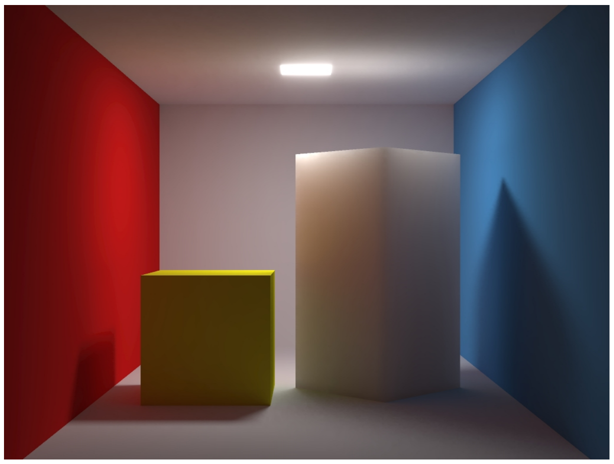


Figure 4: A global illumination scene with a translucent box. Note the light bleeding through the box, and the color bleeding in the model.

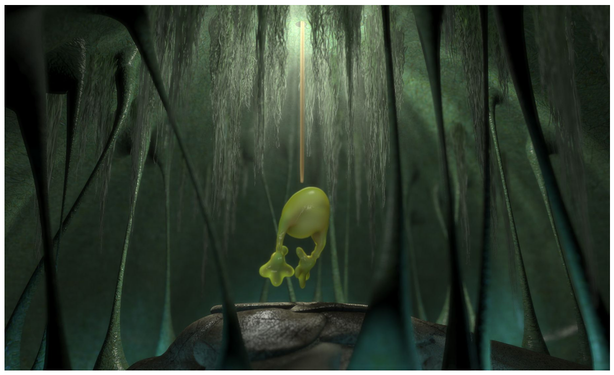


Figure 5: An animation with a translucent character. Translucency helps depict the small size of the character. Image courtesy of Scott Peterson - PDI/DreamWorks.

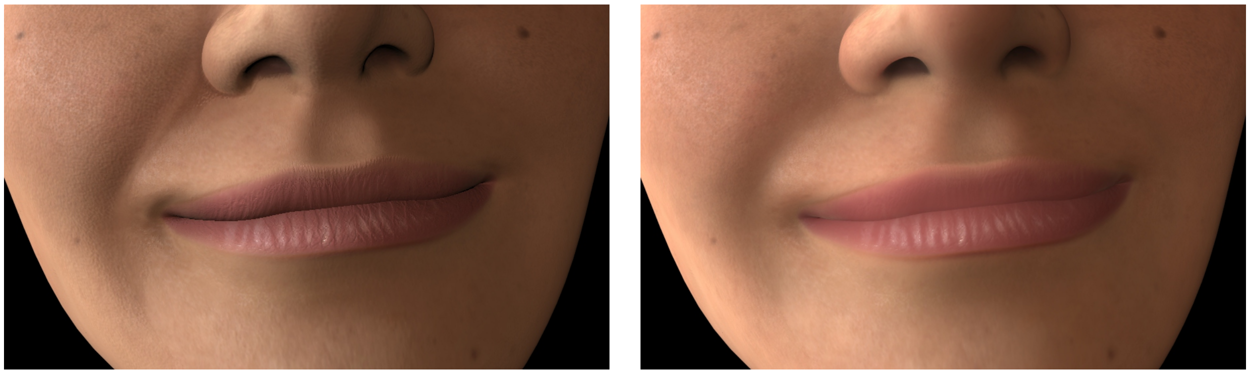


Figure 6: A textured face model lit by three light sources (key, fill, and rim). The left image shows the result using the skin shader that was used in the movie "Shrek", and the right image shows the result after adding our simulation of translucency to this shader.