Volume 23 (2004), number 1 pp. 85–112

COMPUTER GRAPHICS *forum*

Hardware Lighting and Shading: a Survey

Jan Kautz

Massachusetts Institute of Technology, Cambridge, USA

Abstract

Traditionally, hardware rasterizers only support the Phong lighting model in combination with Gouraud shading using point light sources. However, the Phong lighting model is strictly empirical and physically implausible. Gouraud shading also tends to undersample the highlight unless a highly tesselated surface is used. Hence, higher-quality hardware accelerated lighting and shading has gained much interest in the recent five years. The research on hardware lighting and shading is two-fold. On the one hand, better lighting models for local illumination (assuming point light sources but evaluated per pixel) were demonstrated to be amenable to hardware implementation. On the other hand, recent research has demonstrated that even area lights, represented as environment maps, can be combined with complex lighting models. In both areas, many articles have been published, making it hard to decide which algorithm is well-suited for which application. This state-of-the-art report will review all relevent articles in both areas, and list advantages and disadvantages of each algorithm.

Keywords: real-time rendering, lighting and shading, graphics hardware, BRDFs, environment maps.

ACM CCS: I.3.1 [Computer Graphics] Graphics processors; I.3.3 [Computer Graphics]: Bitmap and frame buffer operations; I.3.7 [Computer Graphics] Color, Shading, Shadowing and Texture

1. Introduction

Until the mid-1980s, computer graphics was mainly concerned with offline rendering. Many techniques, such as radiosity and ray tracing were developed to create photorealistic still images, often taking hours to compute.

Yet many applications call for interactive image synthesis. Initial systems could only provide simple wireframe views, but the true potential of interactive 3D graphics soon became apparent. A huge effort was undertaken to improve the speed and quality of interactive techniques. The biggest leap forward was the introduction of hardware support by companies like SGI, which offered high-end workstations with hardware-accelerated 3D graphics. Soon a new direction of research was born: *real-time rendering*. Many new application areas, such as Virtual Reality, benefitted and still benefit from the advances in this field of research.

Although interactive 3D graphics became more and more commonplace, hardware 3D support could only be found in high-end workstations until the mid-1990s. Then 3D games like Quake appeared, and soon graphics hardware became available for the mass PC market. The first PC 3D graphics cards mainly tried to catch up with developments in the high-end market. After only a few years, at the end of the century, PC graphics started to take the lead. Nowadays low-end PC graphics hardware is capable of handling over 300 million vertices per second, and the fill rate achieves several gigapixels per second. This increase in performance also raises expectations of higher and more realistic image quality. Today, realistic shading is one of the main areas in research on real-time rendering.

Over the recent years, many different algorithms have been proposed in this area. This STAR will present all the related algorithms in a common framework. The advantages and disadvantes of the algorithms will be discussed.

We first present the necessary background in the area of materials and material representation. Then we continue with background on lighting computations and how these computations can be approximated to speed up the process. Before we go on to the actual algorithms, we present the hardware

Submitted July 2003
Revised October 2003
Accepted December 2003

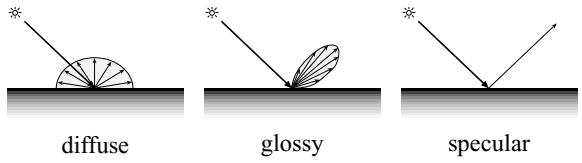

Figure 1: *Different kind of reflections from a single beam of light. Light hitting a surface is reflected in various directions.*

capabilities of current graphics hardware, such that the reader understands the difficulties that need to be overcome to incorporate better lighting and shading in real-time rendering.

2. Materials

First, we would like to show the different kinds of materials that exist in reality. We take a fairly practical approach, and do not classify the materials based on their exact physical properties but rather qualitatively. We then introduce the bidirectional reflectance distribution function, which describes how light is reflected from a surface. It is the most important function for rendering realistic materials.

2.1. Classification

We will classify materials into three categories: opaque, translucent, and transparent. These different classes are not strictly disjoint, but they help understand with which kinds of materials we are concerned.

2.1.1. Opaque Materials

Opaque materials, as the name already suggests, are not translucent at all. This includes for example stone, wood, metals, etc. Light does not penetrate into opaque materials, it is only reflected off the surface (in reality, light may penetrate the material to some extent, but not enough that it needs to be simulated for rendering purposes). Opaque materials can be either homogeneous (optical properties are constant) or heterogeneous (optical properties vary across the surface). A good example of a heterogeneous material is wood, as it exhibits a typical ring structure.

Opaque materials offer various types of reflections. Lambertian *diffuse* reflections are view-independent, i.e. they do not depend on the viewing position. For example, white chalk comes close to exhibiting pure Lambertian reflectance. A reflection is called *specular*, if a material is highly polished, such as a mirror or smooth metals. A *glossy* reflection is called everything between diffuse and specular reflections and is often generated by rough surfaces; varying roughness leads to varying glossiness. A glossy reflection of a light source is often called a *highlight*. See Figure 1 for a depiction of these three kinds of reflections.

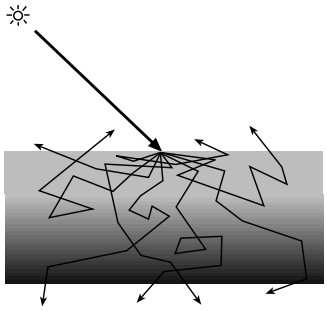

Figure 2: *An example of possible light paths through a translucent material.*

There are two main types of opaque materials: *dielectrics* and *metals*. The highlight of a dielectric has the color of the light source (e.g. plastic, which consists of a white substrate with color pigments that do not contribute to a highlight), whereas metals change the color of the highlight. Furthermore, dielectrics reflect more light at grazing angles than at orthogonal incidence, whereas metals reflect light fairly equally for all angles of incidence. This effect is described by the *Fresnel*-term.

The reflection properties of all opaque materials can be described by a six-dimensional function, which depends on the position on the surface, the incident light direction, and the viewing direction. For more detail see Section 2.2.

Most work on real-time shading is concerned with opaque materials, see Section 6 and Section 7.

2.1.2. Translucent Materials

In contrast to opaque materials, light enters a translucent material, is scattered inside the material and finally leaves the material again. In a translucent material a photon can enter and leave the material at very different positions (called *subsurface scattering*). This creates a very distinct look that is e.g. known from small marble statues but also from skin. For example, when light shines from behind on someone's ear, it will usually look very reddish from the front, since light is scattered inside the blood vessels and exits the ear at the opposite side towards the front, see Figure 2 for an example. Other materials that fall into that category include milk, certain plastics, alabaster, etc.

Conceptually, translucent materials could be represented by an eight-dimensional function, where 2 times 2 dimensions are used for light entrance and exit position, and 2 times 2 dimensions for the incoming and outgoing light direction. However, since this 8D function depends highly on the geometry of the object, no analytical formulation can be used. Instead, the 8D function must be sampled or computed on the fly.

Table 1: *List of used variables/terms*

Variable	Meaning
$\hat{v}$	viewing direction (global)
$\hat{v}_\omega$ or $\hat{\omega}_o$	viewing direction (local)
$\hat{l}$	light direction (global)
$\hat{l}_\omega$ or $\hat{\omega}_i$	light direction (local)
L_e	emitted radiance
L_{in} or L	incident radiance
L_{out} or L_o	reflected exit radiance
$f_r(\hat{l}_\omega, \hat{v}_\omega)$	BRDF
$f_r^*(\hat{l}_\omega, \hat{v}_\omega)$	BRDF product function: $f_r(\hat{l}_\omega, \hat{v}_\omega)(\hat{n} \cdot \hat{l})$
y_i	spherical harmonic functions

Only very recent work [29,53,60] deals with interactive rendering of translucent materials. For instance, precomputed radiance transfer [83] can easily be extended to include subsurface scattering [84].

2.1.3. Transparent Materials

Transparent materials, such as glass, form a special case of translucent materials. Light enters transparent materials but will generally not be scattered inside the material. It will pass through the material and simply exit again.

Nonetheless, transparent materials require special treatment [32,54] since the direction of light changes due to refraction when it enters or exits the material.

2.1.4. Materials covered in this STAR

In this STAR we will only present algorithm for rendering opaque materials. The very recently proposed algorithms for real-time rendering translucent [29,53,60,84] and transparent [32,54] materials will be omitted to narrow the focus of this STAR.

2.2. Bidirectional Reflectance Distribution Function

The bidirectional reflectance distribution function (BRDF) describes how light incident on a surface is reflected into a continuum of directions. It is defined as the ratio of the differential reflected radiance L_o leaving $\underline{x}$ in direction $\hat{\omega}_o$ and the differential irradiance arriving from $\hat{\omega}_i$:

$$f_r(\underline{x}, \hat{\omega}_i \rightarrow \hat{\omega}_o) := \frac{dL_o(\underline{x}, \hat{\omega}_o)}{dE(\underline{x}, \hat{\omega}_i)} = \frac{dL_o(\underline{x}, \hat{\omega}_o)}{L_{in}(\underline{x}, \hat{\omega}_i) \cos\theta_i \, d\hat{\omega}_i}. \tag{1}$$

For a list of variables used, see Table 1. The directions $\hat{\omega}_i$ and $\hat{\omega}_o$ vary over the unit hemisphere and $\underline{x}$ is the 2D position on the surface; the unit of the BRDF is $[1/sr]$. It is implicitly assumed to depend on the wavelength λ as well, i.e. the BRDF is possibly different at different wavelengths λ (often, it is only defined for the RGB color channels separately).

The BRDF can be extended to account for refraction and transmission (in case the material is translucent), it is then called the *bidirectional scattering distribution function* (BSDF). In this case the directions $\hat{\omega}_i$ and $\hat{\omega}_o$ vary over the unit sphere.

If the BRDF depends on the position $\underline{x}$, it is often called *spatially varying* or *shift-variant*, otherwise the material is homogeneous. The dependency on $\underline{x}$ is often implicitly assumed, removing the parameter $\underline{x}$.

A material is called *anisotropic* if its reflection changes when the surface is rotated about the normal. Otherwise it is called *isotropic*. Let $\hat{\omega}_i = (\theta_i, \phi_i)$ and $\hat{\omega}_o = (\theta_o, \phi_o)$, where θ is the elevation and ϕ is the azimuth, then the BRDF for isotropic materials simplifies to $f_r(\underline{x}, \theta_i, \theta_o, \phi_o - \phi_i)$, dropping one dimension.

Although the BRDF in its general form is already a six-dimensional function (plus another dimension for the wavelength dependency), it already makes a few assumptions [25].

First, it assumes that light exits the surface at the same location where it hit the surface. This means that certain surfaces, such as marble, cannot be correctly modeled with a BRDF, since those materials exhibit a considerable amount of *subsurface scattering*.

The definition of the BRDF further implies that light is reflected immediately, i.e. light is not stored for some time and then re-emitted later (*phosphorescence*).

Finally, the BRDF cannot model materials that change the frequency of the incident light (*flourescence*). For example, neon colors cannot be represented, since they absorb light at certain wavelengths and re-emit light at different wavelengths, making the color appear overly bright.

The BRDF has to fulfill two important properties to be physically plausbile. First it needs to be energy conserving, i.e. no more energy must be reflected than is received [7]. Furthermore, it must obey the *Helmholtz reciprocity* [7], meaning that the BRDF must be symmetric in $\hat{\omega}_i$ and $\hat{\omega}_o$. Please note that this only holds for reflections but not for refractions, i.e. not for the BSDF.

3. Rendering Techniques

In this section, we will outline how the illumination in a scene can be solved in a general way and how its results are displayed. We will explain in more detail how approximations can be made, so that graphics hardware can be exploited to speed up the computation.

3.1. Rendering Equation

The complete illumination of a scene (neglecting effects such as subsurface scattering or participating media, flourescene

and phosphorescence) is usually described using the rendering equation proposed by Kajiya [40]. It is an integral equation describing all the light exchange in a scene:

$$L_o(\underline{x}, \hat{v}) = L_e(\underline{x}, \hat{v}) + \int_{\Omega^+} f_r(\underline{x}, \hat{l}_\omega, \hat{v}_\omega) L_{in}(\underline{x}, \hat{l})(\hat{l} \cdot \hat{n})\, d\hat{l}, \tag{2}$$

where $\hat{v}$ is the viewing direction and $\hat{l}$ is the light direction in world-space, $\{\hat{n}, \hat{t}, \hat{n} \times \hat{t}\}$ is the local coordinate frame of the reflective surface, $\hat{v}_\omega = \omega(\hat{v}, \hat{n}, \hat{t})$ represents the viewing direction and $\hat{l}_\omega = \omega(\hat{l}, \hat{n}, \hat{t})$ the light direction relative to that frame, f_r is the BRDF, which is usually parameterized via the local viewing and light direction.

This equation says that the radiance leaving $\underline{x}$ towards $\hat{v}$ (e.g. the direction towards the viewer) equals the radiance emitted from $\underline{x}$ in direction $\hat{v}$, in case $\underline{x}$ lies on an emitter, plus all the reflected radiance, i.e. the integral over all the incident radiance at $\underline{x}$ scaled by the BRDF and the cosine weighting term.

This equation also accounts for *indirect illumination*, since the incident radiance L_{in} possibly includes radiance reflected from other parts of the scene (i.e., $L_{in}(\underline{x}, \hat{l}) = L_o(\underline{y}, -\hat{l})$).

The rendering equation can also be expressed as an integral operator, which takes a function of incident radiance and generates exit radiance for given viewing direction. The linear operator $\mathcal{R}$ is defined as:

$$(\mathcal{R}L_{in})(\underline{x}, \hat{v}) = \int_{\Omega^+} f_r(\underline{x}, \hat{l}_\omega, \hat{v}_\omega) L_{in}(\underline{x}, \hat{l})(\hat{l} \cdot \hat{n})\, d\hat{l}. \tag{3}$$

All the methods presented in this article implement (or approximate) this operator. This implies that the viewing direction as well as the incident lighting are allowed to change. Other real-time rendering methods are not included in this STAR.

3.2. Accurate Solutions of the Rendering Equation

There are two main approaches that can be used to solve the rendering equation. The first approach traces individual rays to solve it, whereas the second approach determines the energy transfer between surface patches until an equilibrium is reached. Although these methods are accurate in a sense that they do not directly simplify the rendering equation, they still might introduce bias or not support all types of transport paths, e.g. caustics.

In this STAR, we are concerned with real-time shading and lighting using graphics hardware, so we list these methods for completeness only.

3.2.1. Ray Tracing Methods

As the name already implies, ray tracing methods trace rays (transferring radiance) through a given scene in order to compute the light transport between all components.

Many variants of ray tracing have been developed to solve the rendering equation, e.g., path tracing [40], bidirectional path tracing [50], distribution ray tracing [17], photon mapping [38], density estimation [81], and many more.

Recently, it has been shown that ray tracing can also be implemented at interactive rates on PC clusters [90,89] and even on commodity PC graphics hardware [75]. However, high frame rates do require many PCs, whereas hardware-accelerated rendering can achieve realtime rates even on a single PC.

3.2.2. Radiosity

Radiosity [15] methods subdivide the entire scene geometry into patches. Some of the patches are emitters, whereas the other patches are receivers. The rendering equation assumes that the illumination is in equilibrium (emitted photons equal the number of absorbed photons), so energy is exchanged between patches until the solution converges (of course taking visibility into account).

Early radiosity methods work only for diffuse receivers but can also be extended to glossy receivers [37,15]. The result is usually stored at the vertices of the subdivided geometry.

Research on radiosity has decreased, since Monte Carlo techniques, such as photon mapping [38], seem to be better suited to accurately solving the rendering equation (although for purely diffuse scenes it converges slowly). Recently, it was shown that photon mapping can even be implemented on graphics hardware [74], but rendering takes still a couple of seconds for small scenes and few photons.

3.3. Approximate Solutions of the Rendering Equation

The rendering equation can be simplified in order to be computed more quickly. These approximations also make it more amenable to graphics hardware. Here we list popular choices of approximate solutions.

3.3.1. Ambient Illumination

If the algorithm used for image synthesis does not compute indirect illumination, an ambient term is usually introduced which tries to account for all the indirect illumination re-emitted from all surfaces. It cannot be expressed as a BRDF, it is simply the average emitted radiance L_a in the scene scaled by a constant k_a, which is added to the rendering equation:

$$L_o(\underline{x}, \hat{v}) = k_a L_a + L_e(\underline{x}, \hat{v}) + \int_{\Omega^+} f_r(\underline{x}, \hat{l}_\omega, \hat{v}_\omega)\ \ L_{in}(\underline{x}, \hat{l})(\hat{l} \cdot \hat{n})\, d\hat{l}, \tag{4}$$

where we assume that the incident illumination L_{in} only accounts for *direct illumination*, i.e. light emitted directly from a light source.

This new equation can be solved more easily. For every point $\underline{x}$, it is only necessary to integrate the light incident from light sources that are not blocked on the way to $\underline{x}$. This can be written more clearly by expanding the incident radiance into two terms

$$L_{in}(\underline{x}, \hat{l}) = L_s(\underline{x}, \hat{l})V(\underline{x}, \hat{l}), \quad (5)$$

where the term L_s is the radiance received from light sources, and the term V takes care of self-shadowing or shadowing from other objects.

3.3.1.1. Ambient Occlusion A more accurate ambient term takes the visibility at $\underline{x}$ into account [98,14]:

$$L_{amb}(\underline{x}) := k_a L_a \int_{\Omega^+} V(\underline{x}, \hat{l})(\hat{l} \cdot \hat{n})\, d\hat{l}. \quad (6)$$

This is usually called *ambient occlusion*. The advantage of using this ambient term is that occluded areas of an object appear darker. This is exactly what an observer expects and hence images rendered with this technique look more realistic.

3.3.2. Distant Illumination

Illumination of an object can often be simplified by assuming that the incident illumination is at infinity in addition to assuming no interreflections (we will also assume the object does not emit light without limiting generality). The incident illumination term then simplifies to:

$$L_{in}(\underline{x}, \hat{l}) = L_s(\hat{l})V(\underline{x}, \hat{l}), \quad (7)$$

dropping the dependency on $\underline{x}$. This approximation is commonly used in interactive computer graphics, where $L_s(\hat{l})$ is represented using an *environment map* [10]. We will refer to this case as *global illumination* for real-time rendering, although it does not include all effects that are usually associated with global illumination [17,40,15], such as indirect illumination and caustics.

3.3.2.1. Without Shadowing. If shadows are neglected then the original rendering equation simplifies to:

$$L_o(\underline{x}, \hat{v}) = \int_{\Omega^+} f_r(\underline{x}, \hat{l}_\omega, \hat{v}_\omega)L_s(\hat{l})(\hat{l} \cdot \hat{n})\, d\hat{l}. \quad (8)$$

For special kinds of BRDFs it has been shown that this equation can be precomputed [62,27,11,35,42,77,51,58], but only if the BRDF is not spatially varying and obeys certain restrictions. Rendering is then just the application of a *prefiltered* environment map. On-the-fly filtering is also possible if certain restrictions apply [45,77,83]; spatially varying materials can also be simulated [45]. We will detail these techniques in Section 7.

3.3.2.2. With Shadowing As you can see in the following equation

$$L_o(\underline{x}, \hat{v}) = \int_{\Omega^+} f_r(\underline{x}, \hat{l}_\omega, \hat{v}_\omega)L_s(\hat{l})V(\underline{x}, \hat{l})(\hat{l} \cdot \hat{n})\, d\hat{l}, \quad (9)$$

shadowed distant illumination is fairly complicated to compute, since the shadowing term varies for every $\underline{x}$. Nonetheless, real-time evaluation of this integral is possible as shown by Sloan et al. [83]. This technique can illuminate static objects with distant (low-frequency) lighting **including** self-shadowing and even interreflections at interactive rates. Recently Ng et al. [69] showed that shadowing can also be incoporated for distant higher-frequency lighting as well.

These techniques will be presented in detail in Section 7.

3.3.3. Point Lights

Illuminating a scene with point lights greatly simplifies the computations, especially since interreflectons are commonly ignored as well in this case. The rendering equation then simplifies to:

$$L_o(\underline{x}, \hat{v}) = k_a L_a + \sum_{j=0}^{n} f_r(\underline{x}, \hat{l}_\omega, \hat{v}_\omega)\frac{I_j}{r^2}V(\underline{x}, \hat{l})(\hat{l} \cdot \hat{n}), \quad (10)$$

where I_j is the intensity of light source j (assuming it has a uniform spherical distribution), and r is the distance from the light source to $\underline{x}$. As you can see, the only complicated part that is left is shadowing and the BRDF. We will refer to this case as *local illumination*.

3.3.3.1. Without Shadowing If even shadowing is neglected, then only the BRDF has to be evaluated at every visible point of the scene. Traditionally, graphics hardware only handles this case. Even more, it only directly supported the Blinn-Phong BRDF model [9].

Hence, much research has focused on including more complex reflectance models into real-time rendering [35,41,44,57,59,58,65]. A detailed overview of these methods will be presented in Section 6.

3.3.3.2. With Shadowing Shadowing for point light sources can be seen as an extension to the unshadowed case. We can first shade every point in the scene ignoring shadows, and then in a second pass check which points are actually in shadow and darken (corresponds to assuming some ambient illumination) or blacken these points.

There are two main techniques using either *shadow volumes* [18] or *shadow maps* [93]. Both these methods can be accelerated using graphics hardware. Recent research deals with linear light sources [30] and with spherical light sources [1] which cast soft shadows.

This STAR is not mainly concerned with shadowing from point light sources, but the most important algorithms will be explained in Section 6.4.

3.4. Displaying the Solutions

Some of the above mentioned algorithms for solving the rendering equation (e.g. radiosity) still need an additional rendering pass to actually display the solution, since the solution is only stored on the surfaces. There are two main display methods: one is again based on ray tracing, and the other method is based on (hardware-accelerated) rasterization.

3.4.1. Ray Tracing

The ray tracing approach traces rays from the camera through every pixel on the viewplane into the scene. At every intersection point, it queries the stored solution and displays the result of the query. The query depends on the strategy that was used to compute the solution of the rendering equation. E.g., for a diffuse-only radiosity algorithm this corresponds to looking up the stored radiosities at the vertices of the intersected patch, computing a filtered result, and converting it to exit radiance.

3.4.2. Rasterization

Rasterization is what graphics hardware usually does to render the geometric primitives (see also next section), but it can also be implemented in software.

Rasterization iterates over all primitives and renders each primitive into a *framebuffer* according to the current camera settings. Rasterization first projects a primitive (usually only triangles) to its 2D screen coordinates. Then it iterates over all pixels that the projected primitive takes up in screen-space; for every pixel the radiance value (interpolated from the vertices or by texture lookup) and the depth is stored in the framebuffer. During rasterization of a primitive, its depth value at the current pixel position is compared to what has already been stored at that position, and only if it is in front of the old content, it is rendered.

This can also be used to display a radiosity solution. At every vertex we look up the stored radiosity, convert it to radiance, set it as the color at the vertex, and then just rasterize the primitive. For every pixel in the framebuffer we will get the bilinear filtered radiance, correctly taking occlusions into account.

In the next section, we will take a closer look at graphics hardware.

4. Hardware Rendering Pipeline

In this section we look at current graphics hardware, how it works and what features it supports. Graphics hardware is accessed via a graphics API, such as OpenGL [79,68] or DirectX [61]. In this work we discuss OpenGL only but DirectX could be used as well, as both APIs offer similar functionality.

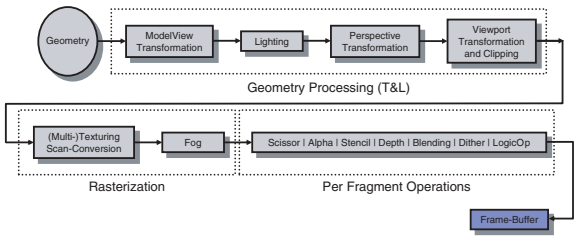

Figure 3: *The standard fixed function rendering pipeline.*

Most graphics hardware implements a variation of the standard rendering pipeline [24]. Figure 3 depicts the pipeline as it is defined by OpenGL. Geometry is sent to the graphics hardware—usually only polygons, lines, and points are supported. It is first processed in the *geometry processing* stage, also called *transform and lighting (T&L) unit*, which transforms the 3D geometry and also performs the lighting computations. Then during the *rasterization* stage, the projected geometry is scan-converted and textured, and then fog is added.

The created *fragments* are piped through a series of tests (e.g. alpha test, depth test, ...), which they can pass or fail. Finally the fragments are blended with the already stored fragments, and the result is written to the *framebuffer*.

This pipeline is also known as the *fixed function pipeline*, since the functions executed for every incoming geometric primitive are fixed in order, they can only partially be turned on or off, or modified to some degree. Newer graphics hardware [70,65] supports a modified pipeline that is more programmable [54].

In the following, we will take a closer look at the individual units of the hardware rendering pipeline.

4.1. Geometry Processing

The main task of the T&L unit, which usually works with floating point data, is to transform the geometric primitives according to some specified transformation matrices. Geometry is specified in terms of *vertices* (given in homogeneous coordinates). Each vertex is first transformed with the *modelview* matrix from object coordinates into the viewing coordinate system. Normal vectors (needed for lighting) are transformed by the inverse transpose of the modelview matrix and are optionally renormalized.

After a vertex has been transformed, lighting computations (with point or directional light sources) are performed. The fixed function pipeline graphics hardware only supports one lighting model, the Blinn-Phong model [9], which is simple to compute but unfortunately fairly limited.

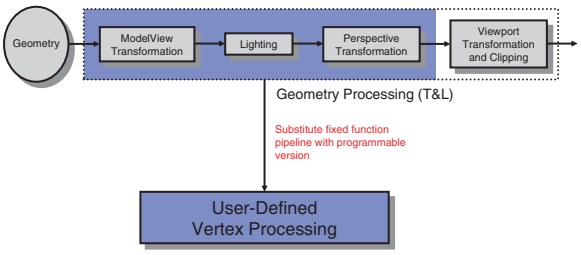

Figure 4: *The new vertex shader functionality replaces parts of the standard fixed function rendering pipeline.*

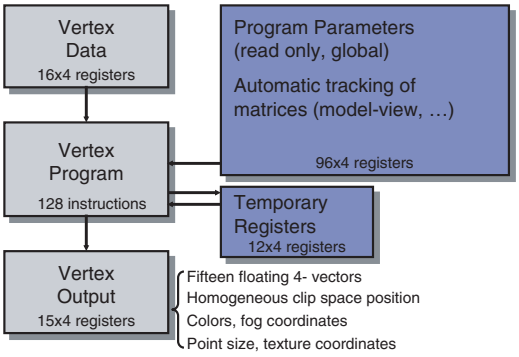

Figure 5: *Vertex shader overview. The mentioned numbers are for a GeForce 3.*

After the lighting computations, the vertices are transformed with the *perspective* matrix, which maps from view coordinates to device normalized coordinates. All primitives are now clipped and then the vertices are transformed into screen coordinates using the viewport transformation.

Texture coordinates are also specified together with vertices and normals. They are represented in homogeneous coordinates as well to allow for projective textures. Texture coordinates can also be generated automatically in the T&L unit, either by a linear combination of the vertices' coordinates or with special modes for environment mapping [28].

In the last two years, it became clear that this fixed pipeline cannot accommodate all the needs of 3D programmers. Lighting calculations were not general enough, more complex lighting models were desirable. Furthermore the texture coordinate generation was very limited. As a result, programmable geometric processing units were developed [54].

4.1.1. Vertex Shader

The programmable part of the T&L unit is called *vertex shader* or *vertex program*. It completely substitutes the first three stages of the fixed T&L pipeline, see Figure 4, which also means that a vertex program always has to implement all three stages of the fixed function pipeline, unless one stage, e.g. lighting, is not needed by the application.

A vertex shader is an assembler program that runs on the graphics card. A vertex shader gets an untransformed, unlit vertex, possibly including normals, colors, and other data as its input, from which it creates a transformed vertex. Optionally, it can also compute lighting at the vertex, create texture and fog coordinates, and also change the point size for point primitives. Current vertex shaders, however, **cannot** create or delete a vertex or change topology, nor can they access data from more than one vertex at once.

The instruction set is tailored towards vertex processing. It is a SIMD instruction set, where each instruction works on 4-floats (e.g. colors or coordinates). The initial version of the vertex shaders [54] supports almost 20 instructions, including instructions for computing dot products, reciprocals, and even logarithms. The instruction set supports input and output mappings. For example, the input can be negated or its components can be swizzled (i.e. arbitrarily rearranged). The output can be written specifically to certain components of the 4-floats only. The latest version of vertex shaders even allows some simple branching [65], but for increased performance set-on-comparison style operations are also supported.

Automatic tracking of global matrices, such as the modelview and perspective matrix, is also supported. In Figure 5, one can find an overview of vertex shaders (specifications for an NVIDIA GeForce 3). Note that storage and input/output registers are limited and register spilling is not possible.

4.2. Rasterization

After the T&L unit has transformed each primitive, their associated data, i.e. transformed vertices, associated colors and texture coordinates are passed on to the rasterizer. The rasterizer scan-converts these primitives yielding *fragments*. A fragment consists of depth, color, alpha value, and texture coordinates; it can be seen as a preliminary pixel that still has to undergo the per-fragment operations; see Section 4.3. Scan-conversion interpolates the incoming data linearly, except for the texture coordinates, which are interpolated in a perspectively correct manner.

If texturing is enabled, the rasterizer does a lookup into the specified (one to three-dimensional) texture map at the interpolated coordinates (with multi-texturing the lookup can be done into multiple textures at the same time). The color retrieved from the texture is then blended together with the interpolated vertex color (for multi-texturing the results from the multiple textures are blended iteratively). Different blending modes are available, e.g. multiplication and addition, as well as compositing operators [68,79].

Rasterization is usually done in fixed-point arithmetic, on lower-end systems with 8 bits and on high-end systems such

as the SGI Onyx with 12 bits. Newer graphics hardware, such as the ATI Radeon 9700 or NVIDIA's GeForce FX even support floating point arithmetic in the rasterizer (although not for blending operations).

4.2.1. Fragment Shader

The programmable *fragment shader* or *pixel shader*, replaces the old (multi-)texturing units. It provides a similar functionality as the vertex shader. A small assembler program is executed on the graphics card, but this time for every fragment instead of every vertex.

Initial versions of the pixel shader, as in NVIDIA's GeForce series, where it was called the *register combiner*, provided a reduced level of programmability. We will rather summarize the new functionality now available in ATI's Radeon 9700/9800 and NVIDIA's GeForce FX.

As just explained, the fragment shaders execute a user-defined program. The input to the program is the interpolated color value, the texture coordinates, and also user-defined data. The instruction set works on 4-vectors (color and alpha); operations include dot-products, multiplications, etc. More complicated operations such as reciprocals and square roots are also available [61,65].

Texture lookups are also very flexible with fragment shaders. Texture access is possible at different places in the fragment shader, and not necessarily only at the beginning of a shader. The most interesting operation is the *dependent texture lookup*. In this case, texture coordinates are computed **in** the fragment shader itself and then the lookup is performed in the same shader. This for example allows us to sample complex functions into textures, and then to do a lookup into the texture instead of evaluating the complex function. Various other applications have already been shown [36]. Fragment shaders do not yet support branching, again for performance reasons (SIMD execution), although upcoming hardware may include such capabilities.

The latest hardware [65,57] works with 16, 24 or 32 bit floating point numbers, up from 16 bit fixed point on the ATI Radeon 8500, and 8 bits on the NVIDIA GeForce cards.

4.3. Per-Fragment Operations

A number of tests, if enabled, are performed before a fragment is written to the framebuffer. We will only name the commonly used ones.

The *alpha test* tests the fragment's alpha value against a user-specified reference value. The *stencil test* compares a reference value against the stencil value stored in the *stencil buffer* at the fragment's position. Depending on the result the stencil buffer is modified and the fragment passes or fails. Finally, the fragment's depth value is tested against the depth value stored in the framebuffer.

Fragments passing all tests are written to the framebuffer. The color and alpha values of a fragment can either be combined with the values already stored in the framebuffer (*blending*) or directly written to the framebuffer.

4.4. Framebuffer

The *framebuffer* consists of several separate buffers. The *color buffer* stores the color and the alpha value of a pixel. The *depth buffer* stores the depth at a pixel. Stencil values are stored in the *stencil buffer*. The color buffer often has only 8 bits per component per pixel, which comes to 32 bits total. On most graphics hardware, the depth buffer has at least 24 bits. The stencil buffer usually only has 8 bits and is often combined physically with the depth buffer.

When a frame is finished, i.e. after it has been rendered, the color-content of the color buffer is displayed. Whatever is stored in the alpha channel or in the other buffers is not displayed; it is only needed during rasterization.

4.5. Per-Pixel vs. Per-Vertex

Graphics hardware is now programmable at the vertex and at the fragment level. So one has to decide which parts of an algorithm should be implemented where.

It is preferable to perform computations at the pixel level that involve quickly varying input data or that produce quickly varying results. For example, it is better to compute glossy lighting at the pixel level instead of only at the vertices (with bilinear filtering across a primitive), since artifacts may arise from undersampling the lighting and highlights might be completely missed. On the other hand, slowly varying data can be easily computed per vertex, for example diffuse lighting.

Of course, this is also a quality/performance trade-off. Higher quality can be achieved with per-pixel computations while using only per-vertex computations might be faster.

4.6. Summary

Current graphics hardware, and future graphics hardware even more, is very flexible and, as it turns out, well-suited to perform complex tasks for which it was not even designed for.

The work presented in this STAR makes heavy use of the new functionality. We will not give very detailed information about the actual implementations, since for newer or different hardware the implementation will change anyway. More information is only given if it is necessary to understand occurring problems or artifacts.

5. STAR Overview

We will first cover all algorithms concerning local illumination (Section 6). We will start with rendering homogeneous

materials, go on to heterogeneous materials, and finally present how shadowing and interreflections can be simulated.

Then, in Section 7, we present all methods for incorporating global incident lighting (based on environment maps). We first start with techniques that need to filter the incident lighting in a preprocess and are therefore only useful for static incident lighting. Then we present techniques that allow to change the incident lighting on-the-fly. Finally, we review recent work on incorporating self-shadowing and interreflections for globally lit objects.

At the end of both sections, we will compare the presented algorithms according to a set of common criteria. This comparison is summarized in Figure 11.

6. Local Illumination

In this section we will deal with local illumination, i.e. with illumination from point light sources. We present a range of real-time shading methods for this specific case.

The reason why many algorithms deal with this special case can be seen in Equation 10. Point lights are much easier to handle than global illumination, because no integral is necessary for computing the visible exit radiance.

6.1. Blinn-Phong (Standard OpenGL)

Standard OpenGL only supports one specific BRDF, the Blinn-Phong model [9], which is defined as follows:

$$f_r(\hat{l}, \hat{v}) = \frac{k_d}{\pi} + \frac{k_s}{(\hat{n} \cdot \hat{l})}(\hat{h} \cdot \hat{n})^N, \qquad (11)$$

where $\hat{l}$ is the light direction, $\hat{v}$ the view direction, and $\hat{h}$ the halfway vector between light and view direction. k_d is the diffuse coefficient, k_s the specular coefficient and the exponent N defines the shininess of the material.

This model is neither reciprocal nor energy conserving, but its mathematical simplicity allowed to build graphics hardware that can evaluate it directly. Unfortunately, standard OpenGL only evaluates the model at every vertex and uses bilinear interpolation within triangles (also called *Gouraud* shading). For mostly diffuse surfaces this is fine, but for more specular objects, this often results in undersampled highlights. These problems lead to extensive research in hardware-accelerated shading. Some of this research deals with incorporating homogeneous BRDFs and other research with inhomogeneous BRDFs.

6.2. Homogeneous BRDFs

In this part of the STAR we will explain how surfaces with homogeneous materials can be rendered at interactive rates.

6.2.1. Analytical Shaders (per Vertex)

Since today most graphics hardware supports vertex shaders, and since many mathematical operations are supported, some BRDFs can be implemented directly in such a vertex shader. NVIDIA [71] has shown that e.g. the Minnaert BRDF [64] can be implemented per vertex. This technique can also produce Gouraud artifacts, if the mesh used is not tesselated finely enough. Hence, this method is not commonly used, but works on older generation cards, since vertex shader can be simulated on the host.

6.2.2. Analytical Decomposition

Heidrich and Seidel [35] proposed analytical decompositions of BRDFs so they can be incorporated into real-time rendering. We will use the Cook-Torrance BRDF model [16] to illustrate the technique, as used by Heidrich and Seidel [35].

Consider the Cook-Torrance BRDF model [16]:

$$f_r(\hat{v}, \hat{l}) := \frac{F(\hat{h} \cdot \hat{l})G(\hat{n} \cdot \hat{l}, \hat{n} \cdot \hat{v})D(\hat{h} \cdot \hat{n})}{\pi(\hat{n} \cdot \hat{l})(\hat{n} \cdot \hat{v})}, \qquad (12)$$

where $F()$ is the Fresnel term, $G()$ the geometric term [85], and $D()$ is the micro-facet distribution [7]. Plugging the above equation into the local radiance equation (Equation 10) one can see that the term $(\hat{n} \cdot \hat{l})$ from the BRDF's denominator cancels out. Now we see that the term $F(\hat{h} \cdot \hat{l})D(\hat{h} \cdot \hat{n})$ only depends on two variables and that the term $G(\hat{n} \cdot \hat{l}, \hat{n} \cdot \hat{v})/(\hat{n} \cdot \hat{v})$ also only depends on two variables. Heidrich and Seidel then tabulate these two terms into two two-dimensional texture maps. 2D texture mapping is used to implement the lookup proecss. During rendering, one only has to compute the texture coordinates for the textures (simple dot-products, software or vertex shader). The results from both textures need to be multiplied together, which can be done using blending or in a pixel shader. This way the reflectance model can be changed from the Blinn-Phong to the physically plausible Cook-Torrance model. Furthermore, evaluation is now done per-pixel avoiding undersampling artifacts from standard OpenGL.

The same technique can be applied to other models, such as the anisotropic Banks model [5], which was also shown by Heidrich and Seidel [33]. Two examples are found in Figure 6. The Ashikhmin BRDF [3] can also be analytically decomposed as shown by Steigleder [86].

6.2.3. Numerical Decomposition

A separable decomposition approximates a given 4D BRDF (i.e. homogeneous) through numerical decomposition, e.g. using singular value decomposition [43] or homomorphic factorization [59].

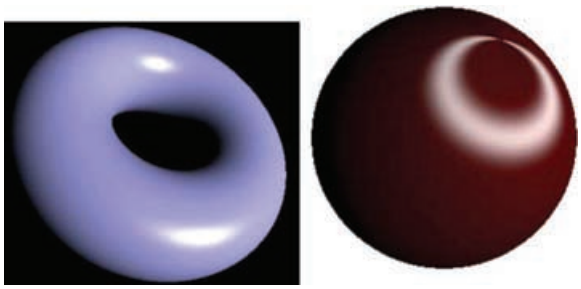

Figure 6: *Two examples for analytical decomposition* [33,35]. *Left: Cook-Torrance model* [16]. *Right: Banks model* [5].

Each of the two directions that a BRDF uses can be modeled as a 2D parameter, hence a reflectance model usually depends on 4 parameters. For an accurate representation this 4D function could just be sampled, but graphics hardware does not support 4D textures directly and a lot of memory would be needed for this representation.

Instead a separable decomposition is used, which approximates the 4D function with a sum of products of two 2D functions:

$$f_r(\hat{v}_\omega, \hat{l}_\omega) = \sum_i g_i(\hat{v}_\omega) h_i(\hat{l}_\omega),$$
$$L(\hat{v}_\omega) = \sum_i g_i(\hat{v}_\omega) h_i(\hat{l}_\omega) \frac{I}{r^2} (\hat{n} \cdot \hat{l}_\omega)$$

As it turns out, for many materials only a single product between two functions is needed, simplifying the computation.

Using texture mapping, $L(\hat{v}_\omega)$ can be easily evaluated on the graphics hardware. Each of these 2D functions $g(\hat{v}_\omega)$ and $h(\hat{l}_\omega)$ (assuming a single-term approximation) can be sampled and stored in a texture map. At every vertex of every polygon, $\hat{v}_\omega$ and $\hat{l}_\omega$ need to be computed and are then used as texture coordinates. Then the polygon has to be texture mapped with the textures containing $g(\hat{v}_\omega)$ and $h(\hat{l}_\omega)$ and the appropriate texture coordinates. Blending has to be set to modulate, so that $g(\hat{v}_\omega)$ and $h(\hat{l}_\omega)$ are multiplied together. Finally, the term $L_{in}(\hat{l}_\omega)(\hat{n} \cdot \hat{l}_\omega)$ has to be multiplied to the result of $g(\hat{v}_\omega)h(\hat{l}_\omega)$, e.g. by enabling OpenGL lighting with only a diffuse component.

Rendering of arbitrary materials using this approximation is very fast because it boils down to computing texture coordinates and blending two texture maps together.

Separating the BRDF along $\hat{l}_\omega$ and $\hat{v}_\omega$ often does not lead to satisfying results. Reparameterizing the original 4D reflectance model in different ways may increase the approximation quality [43,47,96]. See Figure 7 for an example rendered with this technique at real-time rates.

A homomorphic factorization [59] decomposes a function as a product of multiple functions (and not a sum over products between two functions). In the case of BRDFs, a good choice is [59]:

Figure 7: *Hardware accelerated rendering of an anisotropic marble teapot.*

$$f_r(\hat{v}_\omega, \hat{l}_\omega) = p(\hat{v}_\omega) q(\hat{h}_\omega) p(\hat{l}_\omega),$$
$$L(\hat{v}_\omega) = p(\hat{v}_\omega) q(\hat{h}_\omega) p(\hat{l}_\omega) \frac{I}{r^2} (\hat{n} \cdot \hat{l}_\omega),$$

where $\hat{h}_\omega$ is the half-way vector between $\hat{v}_\omega$ and $\hat{l}_\omega$. The factorization of $f_r(\hat{v}_\omega, \hat{l}_\omega)$ is computed in log-space, which transforms this problem into a simple linear data-fitting problem. The results achieved using this decomposition are better than the approximation using singular value decomposition [59].

Recently, Suykens et al. [88] proposed chained matrix factorization, which decomposes a function as a sum of products of multiple functions. It allows the most flexible decomposition. They used it to decompose spatially varying BRDFs data achieving good results.

6.3. Inhomogeneous BRDFs

As mentioned, it is desirable to perform lighting calculations per pixel in order to avoid Gouraud artifacts. In this section we will discuss different possibilities (i.e. bump mapping and arbitrary BRDFs).

6.3.1. Bump Mapping

Blinn [8] has shown how wrinkled surfaces can be simulated by only perturbing the normal vector, without changing the underlying surface itself. The perturbed normal is then used for the lighting calculations instead of the original surface normal. This technique is generally called bump mapping.

If we have another look at the lighting model equation (see Equation 10), we can see a dependence on the normal $\hat{n}$ (i.e.

the local view and light directions are defined relative to it). As mentioned before, OpenGL lighting is usually only evaluated at every vertex and not within a triangle, so normals provided at the vertices are usually used to evaluate Equation 10.

In order to simulate wrinkles, bump mapping requires a per-pixel normal, which is then used when evaluating this equation. Graphics cards now support complex per-pixel operation (see Section 4.2.1) which enables us to perform this bump mapping technique at interactive rates [48].

Bump mapping is fairly simple to implement with these new features. For every pixel we simply have to evaluate the lighting model.

Usually the Blinn-Phong model [9] is used to do bump mapping, because this model mainly uses dot-products. Let us have a look at the lighting equation using the Blinn-Phong reflectance model:

$$L(\hat{v}_\omega) = k_d \frac{I}{r^2}(\hat{n} \cdot \hat{l}_\omega) + k_s \frac{I}{r^2}(\hat{n} \cdot \hat{h})^N$$

If this is used in conjunction with bump mapping, the first term of the sum is usually called diffuse bump mapping and the second term is called specular bump mapping. Using the new per-pixel operations, this formula can be easily computed at every pixel. First, the normals are encoded in a texture map. Then $\hat{\omega}_i$ and $\hat{h}$ are computed on a per-vertex basis (will be interpolated across the triangle). Now, the graphics card has to be configured so it computes the equation above [48].

This method achieves good results and can even be implemented on older consumer graphics hardware [48]. The disadvantage is that on older graphics hardware bump mapping is limited to the Blinn-Phong model.

6.3.2. Bump Mapping with a Spatially Varying Reflectance Model

As just mentioned, bump mapping usually uses the simple Blinn-Phong lighting model [9] for lighting calculations. While this is an appropriate and fast method to do bump mapping, it is not very flexible. The Blinn-Phong model does not have many parameters that can be tweaked to change the appearance of the bumpy surface.

We will introduce a different bump mapping technique [44] which can incorporate (almost) arbitrary analytical BRDF models. It allows to change all parameters of the BRDF on a per-pixel level. See Figure 8 for an example of what can be done.

The main idea is to decompose the BRDF into instructions supported by the fragment shader (e.g. multiplication) and not supported by the shader (e.g. tan()). The unsupported instructions/functions are then tabulated into textures. Using dependent texturing values from the tabulated function can be looked up, effectively evaluating the unsupported function.

Figure 8: *Marble sphere with elevated "veins" using a spatially varying anisotropic Blinn-Phong model.*

This works with a variety of BRDFs, e.g. a modified version of the Blinn-Phong model [44] (see Figure 8), Ward's model [91], or the Lafortune model [49] as shown by McAllister et al. [58].

This method achieves high frame rates, since most BRDFs can be implemented in a single rendering pass. There may be quality problems (see original paper [44]) if quickly varying data (e.g. normals for bump mapping) are looked up from texture maps with only bilinear filtering. This is a general problem if data is stored in texture maps that cannot be interpolated linearly (e.g. vectors). In particular, unit-length interpolated vectors and normals should be renormalized.

6.4. Transfer (Shadowing, Interreflections)

In this section, we will briefly explain various techniques how shadows and interreflections can be incorporated into local illumination.

6.4.1. Standard Shadowing Techniques

There are two main techniques to incorporate shadows for local illumination (see again Equation 10). The first one is called *shadow volumes* [18] and the second one *shadow maps* [93].

Shadow volumes create a volume (using a polygonal representation) around the blocker geometry, which tells you that everything inside this volume is in shadow. Rendering with

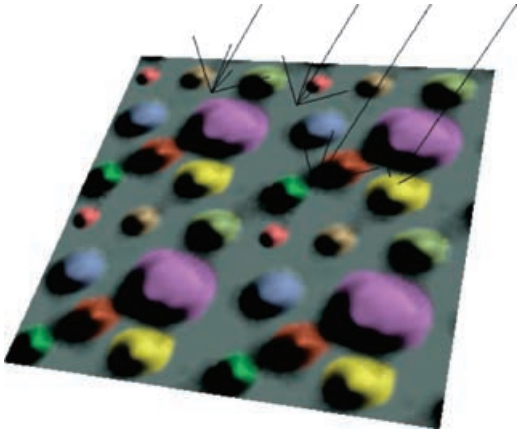

Figure 9: *Simple bump map, where all the bumps are casting shadows according to the light direction.*

shadow volumes can be done quickly by counting how many front- and backfacing polygons defining the shadow volume there are between the pixel to be drawn and the eye point (using the hardware accelerated stencil buffer). However, they require a high fill rate.

Shadow maps work in image space and not in object space. A shadow mapping algorithm renders the scene from the light source and stores the depth values. Then in a second pass, the scene is rendered from the eye point and for every point the algorithm compares the distance to the light source with the stored value. Depending on the result, the point is either in shadow or lit.

Both these methods can be accelerated using graphics hardware. Recent research even deals with linear [30], spherical [1], or almost arbitrary light sources [4].

There are many more papers on shadowing which we will not discuss, as this is not the main focus of this survery.

6.4.2. Shadows in Bump Maps

Bump maps usually do not cast shadows onto themselves, which of course is very unrealistic. We will discuss two techniques [31,82] that add shadows to bump maps. See Figure 9 for an example of a bump map casting a shadow.

Both techniques precompute information on when a pixel is in shadow and store this per-pixel information in texture maps. To decide whether a pixel is in shadow, you only have to know whether the light source position (point or parallel light only) is above or below the horizon visible from that point.

The technique proposed by Sloan and Cohen [82] samples the height of the horizon at a number of position and stores these heights in texture maps. When rendering the bump map they transform the light position into a height value, and use per-pixel operations to perform the comparison between the stored per-pixel height values and the light source height value.

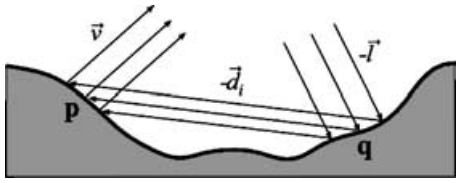

Figure 10: *Light is reflected at point q towards point p and is then reflected towards the viewer.*

Another technique [31] fits an ellipse to the horizon (projected onto the unit disc), stores the parameters of the ellipse in texture maps and performs per-pixel operations to check whether the projected light direction is inside or outside the ellipse, i.e. whether it is lit or in shadow.

Both techniques achieve similar results and comparable rendering speed (single pass rendering in both cases). Only approximate respresentations of the actual horizon are used by both algorithms as well. The approximation quality of Sloan and Cohen's algorithm can be increased more easily by storing more samples of the horizon.

6.4.3. Interreflections

Heidrich et al. [31] showed how interreflections in bump maps can be computed using graphics hardware. This was then extended by Daubert et al. [20] for interreflections in arbitrary geometry. We will explain the basic technique for height fields here.

The fundamental idea of these two methods is to calculate the visibility in a precomputation step, and to store it in a set of scattering textures S_i. In order to do this, a fixed set $D = \{d_i\}$ of sample directions on the sphere is chosen. Then a ray is shot from each grid point in the height field into each of the directions d_i and intersected with the height field geometry. A scattering texture S_i holds the intersection points (as texture coordinates) for all rays starting at any point in the height field in one particular direction d_i. Each of these intersections is uniquely characterized by a 2D texture coordinate.

Let us turn to an example of how the scattering textures are used for computing indirect illumination. Figure 10 depicts the scenario. Light arrives at the height field from direction $\hat{l}$, is reflected at point q in direction $-\hat{d}_i \in D$ and finally leaves the surface in the direction of the viewer $\hat{v}$.

The computation is split into two parts, corresponding to the reflections at q and later at p. First the direct illumination of the height field in viewing direction $-\hat{d}_i$ with light arriving from $\hat{l}$ is computed by a bump mapping step [48,44]

(including self-shadowing, see previous section) and stored in a texture L_d. Afterwards the second reflection is computed in a similar manner. This time the light direction is $\hat{d}_i$ and the viewing direction is $\hat{v}$, however the incoming radiance needs to be looked up in the direct illumination texture L_d. For each surface point p the visible point $q = S_i[p]$ is looked up in the scattering texture corresponding to $\hat{d}_i$. q is used as an index into the direct light texture L_d, yielding the light arriving at p in direction $-\hat{d}_i$. In order to account for interreflections not only from a single point q, the contributions from all $q_i = S_i[p]$ have to be summed up. This algorithm can be easily mapped onto graphics hardware, if dependent texturing is available. The authors report that they achieve interactive results on NVIDIA GeForce cards. However, good quality needs more than 50 sample directions $\hat{d}_i$.

For non-heightfield geometry, the method has to be changed slightly [20]. If the geometry is parameterized, then the intersection points can be uniquely characterized. If it is unparameterized, the method is only applied per vertex, and the vertex number is used for characterizing an intersection (which then have to be quantized to lie on vertices though).

A similar algorithm for interreflections in non-heightfield geometry was proposed by Carr et al. [12]. Their algorithm performs radiosity on the GPU. On the host they precompute the form-factors as well as the links, but the actual radiosity solution is fully computed on the GPU. They successfully apply their technique to subsurface scattering [39] as well.

6.4.4. Combined Self-Shadowing and Interreflections

Polynomial texture maps are also used to perform bump mapping including self-shadowing and interreflections [56]. This technique assumes that the bump map is diffuse only, which makes it view-independent. Hence, the exit radiance at each texel only depends on the light direction. Polynomial texture maps fit a simple polynomial to represent exit radiance for each light direction. All view-independent effects, such as self-shadowing and interreflections, can be directly included in this function. Rendering is straight forward and cheap. A pixel shader has to evaluate the polynomial for the current local light direction.

6.4.5. Bidirectional Texture Functions

Bidirectional texture functions (BTF) were originally introduced by Dana et al. [19]. A BTF stores an *apparent BRDF* [94] at every texel of a texture. An apparent BRDF is similar to a BRDF, but includes non-local effects such as self-shadowing, interreflections and masking effects. Apparent BRDFs are still four dimensional, and hence a BTF is a six-dimensional function. The main issue is how to compress BTFs to reduce storage and also enable real-time rendering.

Several methods for compression and real-time rendering have been proposed. Suykens et al. [88] used the previously mentioned chained matrix factorization. The quality of compression was good, but some details, such as masking effects, were lost. Rendering works in real-time.

Sattler et al. [78] proposed to use PCA on the data. They applied PCA separately for every view direction, resulting in compressed data for every viewing direction. At run-time, data from the closest viewing directions is decoded and interpolated. This yields high quality renderings, but the compression rate is not very high (resulting data is still about 200MB, which is roughly a 1:6 compression ratio).

In contrast to that, Müller et al. [66] applied local PCA, i.e. they first performed k-means clustering and then PCA on each cluster. The resulting compression ratio is about 1:100 and still allows for real-time rendering.

Despite the high quality and realism that can be achieved with BTFs, the high memory requirements limit their application to small tileable patches.

6.5. Comparison

In this section, we classify all local real-time shading algorithms and according to the following categories:

diffuse BRDFs: handles diffuse BRDFs

certain isotropic BRDFs: handles certain fixed isotropic BRDFs

certain anisotropic BRDFs: handles certain fixed anisotropic BRDFs

isotropic BRDFs: handles arbitrary isotropic BRDFs

anisotropic BRDFs: handles arbitrary anisotropic BRDFs

shift-variant BRDFs: BRDFs can vary spatially

bump mapping: supports bump mapping

self-shadowing: supports self-shadowing

interreflections: supports interreflections

dynamic lighting: handles dynamically changing lighting

dynamic objects: handles deforming objects

The actual classification can be found in Figure 11. All the algorithms with an '*l*' are for local illumination and were described in this section. All these algorithm work in real-time. The choice of algorithm still depends on the desired effect. The algorithm by Kautz and Seidel [44] is probably the most flexible and most widely used one. On the other hand only certain BRDFs can be implemented, whereas e.g. other work [41, 59] allows arbitrary but only homogeneous BRDFs

Algorithm	local / global	diffuse BRDF	certain isotropic BRDF	certain anisotropic BRDF	all isotropic BRDFs	all anisotropic BRDFs	shift-variant BRDF	bump mapping	self-shadowing	interreflections	dynamic lighting	dynamic objects
Heidrich and Seidel 1998 [33]	l	x		x							x	x
Heidrich and Seidel 1999 [35]	l	x	x								x	x
Kautz and McCool 1999 [41]	l	x			x	x					x	x
Kilgard 2000 [48]	l	x	x					x	s	i	x	x
Vertex Shader	l	x	x	x							x	x
Heidrich et al. 2000 [31]	l	x	x					x	x	x	x	x
Kautz and Seidel 2000 [44]	l	x	x	x			x	x	s	i	x	x
McCool et al. 2001 [59]	l	x			x	x					x	x
Malzbender et al. 2001 [56]	l	x						x	x	x	x	x
McAllister et al. 2002 [58]	l	x		x			x	p	s	i	x	x
BTF rendering [88, 78, 66]	l	x			x	x	x	x	x	x	x	x
Heidrich and Seidel 1999 [35]	g	x	x					x				x
Kautz and McCool 2000 [42]	g				x							x
Cabral et al. 1999 [11]	g	x			x							x
Latta and Kolb 2002 [51]	g	x			x			p				x
McAllister et al. 2002 [58]	g			x			x	p				x
Ramamoorthi and Hanrahan 2001 [76]	g	x									x	x
Ramamoorthi and Hanrahan 2002 [77]	g	x			x						n	x
Kautz et al. 2000 [46]	g				x			p			x	x
Kautz et al. 2002 [45]	g	x			x	x	x				x	x
Sloan et al. 2002 [83]	g	x			x	x	x	d	x	x	x	w
Lehtinen et al. 2003 [52]	g	x			x	x	x		x	x	x	w
Sloan et al. 2003 [84]	g	x			x	x	x		x	x	x	w
Ng et al. 2003 [69]	g	x					x	p	x	x	x	w

Figure 11: ***Classification of Algorithms***. *l = local, g = global, x = supported, p = potentially, but not described in paper, s = shadowing with* [31], *i = interreflections with* [31,82], *n = near interactive, d = diffuse in real-time, otherwise interactive, w = only without shadowing/interreflections.*

(although one can always render the same objects several times with different homogeneous BRDFs and use an alpha texture to blend between them, effectively creating a non-homogeneous material). BTFs generate impressive results, but at the cost of high storage requirements, which limits the number of different materials that can be used in a scene.

7. Global Illumination

In this section, we will explain relevant techniques for real-time global illumination. Here, global illumination refers to globally incident light as described in Section 3.3.2. These techniques are mostly based on *environment maps*.

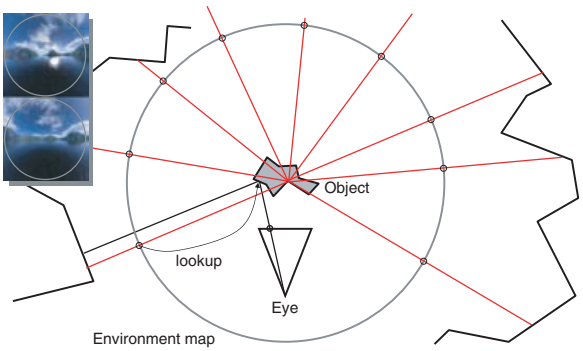

Figure 12: *Radiance incident at a single point is stored in an environment map. Here, we have used the parabolic parameterization* [34] *to store the incident radiance.*

7.1. Environment Maps

Blinn and Newell [10] first introduced the *environment map* technique for producing mirror-like reflections on curved objects. An environment map stores the radiance incident from all directions at a single point, see Figure 12 for a 2D example. A reflection on an object is created by computing the reflected viewing direction (reflected about the surface normal) and then using this reflection direction for a lookup into the environment map. Since the environment map is only valid for a single point but a real object has some extent, this technique introduces some parallax error. It basically assumes that the environment is infinitely far away (see Section 3.3.2).

7.2. Filtered Environment Maps

Greene [26,27] observed that a filtered environment map could be used to simulate diffuse and glossy reflections. Instead of storing the incident radiance, Greene simply stored exit radiance, i.e. the incident radiance already integrated against the BRDF. This is the basis of most environment map methods.

Generally speaking, filtered environment maps capture all the reflected exitant radiance towards all directions $\hat{v}$ from a fixed position $\underline{x}$:

$$L_{\text{glossy}}(\underline{x}; \hat{v}, \hat{n}, \hat{t}) = \int_{\Omega} f_r(\underline{x}; \hat{v}_\omega, \hat{l}_\omega) L_{in}(\hat{l})(\hat{n} \cdot \hat{l})\, d\hat{l}, \quad (13)$$

A filtered environment map stores the radiance of light reflected towards the viewing direction $\hat{v}$, which is computed by weighting the incoming light L_{in} from all directions $\hat{l}$ with the BRDF f_r. Note that L_{in} can be interpreted as the unfiltered original environment map. This map should use high-dynamic range radiance values to be physically plausible. In the general case we have a dependence on the viewing direction as well as on the orientation of the reflective surface, i.e. the local coordinate frame $\{\hat{n}, \hat{t}, \hat{n} \times \hat{t}\}$.

However, this general kind of environment map is five dimensional. Two dimensions are needed to represent the viewing direction $\hat{v}$ (a unit vector in world coordinates) and three dimensions are necessary to represent the coordinate frame $\{\hat{n}, \hat{t}, \hat{n} \times \hat{t}\}$; e.g. three angles or a unit quaternion can be used to specify the orientation of an arbitrary coordinate frame.

The prefiltered environment map techniques which we will examine usually drop some dependencies (e.g. on the tangent $\hat{t}$) and are often reparameterized (e.g. indexing is not done with the viewing direction $\hat{v}$, but with the reflected viewing direction).

In this section we will classify diffuse environment maps [62,76], specular environment maps [10], Phong environment maps [35,62], Lafortune environment maps [58], environment maps filtered with isotropic BRDFs [11,42,46,51,77] and anisotropic BRDFs [46,45].

The classification is done separately for environment mapping techniques that work only for static lighting and techniques that can be applied to dynamic lighting.

7.3. Environment Mapping for Static Lighting

We first classify environment mapping techniques that apply to the case of static lighting. Please note that *static* lighting only means that the lighting cannot change completely. However, it is allowed to rotate.

The limitation to static lighting comes from the slow prefiltering process that these methods use. The general filtering idea is depicted in Figure 13. A shift-variant filter kernel (according to some BRDF and the actual proposed technique) is applied to the original environment map containing incident radiance. This yields one pixel in the filtered target environment map. The actual details depend on the chosen method, please see the following sections.

In order to render an object using a prefiltered environment map, we store the prefiltered environment map in a texture (often only 2D). The parameters which the environment map depends on are used as texture coordinates to index into the environment map. Hence, it is necessary to compute the texture coordinates at every vertex (e.g. in a vertex shader). In case of general 5D prefiltered environment maps, one would need to index into the environment map with the view direction and the local tangent frame (total of 5 texture coordinates).

7.3.1. Specular Environment Maps

Specular environment mapping is the traditionally used environment map technique, first introduced by Blinn and Newell

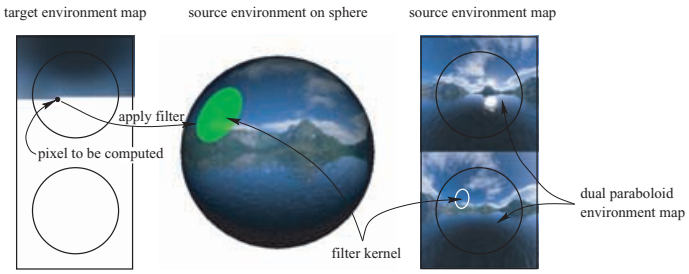

Figure 13: *Filtering of an environment map. A pixel in the target environment map is computed by applying a filter to the source environment map. Both are usually given in a representation like the dual paraboloid map* [34]. *The filter which is defined on the sphere has to be projected to the environment map space.*

[10]. The purely specular BRDF is given by:

$$f_r(\hat{v}, \hat{l}) := \frac{k_s \delta(\hat{r}_v(\hat{n}) - \hat{l})}{(\hat{n} \cdot \hat{l})}, \tag{14}$$

where $k_s \in [0, 1]$ describes the absorption of the surface. The delta Dirac function $\delta(\hat{r}_v(\hat{n}) - \hat{l})$ is infinite when the reflected incident angle $\hat{r}_v(\hat{n}) = 2(\hat{n} \cdot \hat{v})\hat{n} - \hat{v}$ equals the light direction $\hat{l}$. It is defined by the following sifting property: $\int \delta(\vec{a} - \vec{b}) f(\vec{b}) d\vec{b} = f(\vec{a})$.

Now consider Equation 13 using this reflectance function:

$$L_{\text{spec}}(\underline{x}; \hat{v}, \hat{n}, \hat{t}) = \int_\Omega \frac{k_s \delta(\hat{r}_v(\hat{n}) - \hat{l})}{(\hat{n} \cdot \hat{l})} L_{in}(\hat{l})(\hat{n} \cdot \hat{l})\, d\hat{l} \tag{15}$$

$$= \int_\Omega k_s \delta(\hat{r}_v(\hat{n}) - \hat{l}) L_{in}(\hat{l})\, d\hat{l} \tag{16}$$

$$= k_s L_{in}(\hat{r}_v(\hat{n})). \tag{17}$$

Obviously the tangent $\hat{t}$ is not used and can be discarded. Instead of indexing the environment map with $\hat{v}$ and $\hat{n}$, it can be reparameterized so that it is directly indexed by the reflection vector $\hat{r}_v$:

$$L_{\text{spec}}(\underline{x}; \hat{r}_v) = k_s L_{in}(\hat{r}_v), \tag{18}$$

which is what we expect.

An example of a specular environment map can be seen in Figure 14.

7.3.2. Diffuse Environment Maps

Miller [62] has proposed the use of a purely diffuse BRDF to prefilter environment maps. A diffuse BRDF can be written as:

$$f_r(\hat{v}, \hat{l}) := \frac{k_d}{\pi}, \tag{19}$$

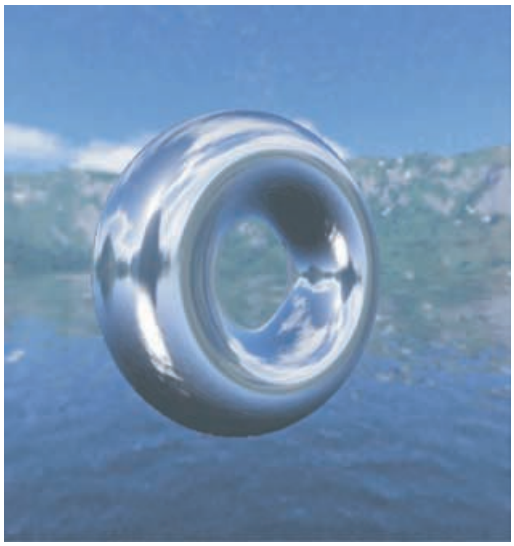

Figure 14: *Torus with specular reflection.*

where $k_d \in [0,1]$ describes the absorption of the surface. Substituting this into Equation 13, we get:

$$L_{\text{diffuse}}(\underline{x}; \hat{v}, \hat{n}, \hat{t}) = \int_\Omega \frac{k_d}{\pi} L_{in}(\hat{l})(\hat{n} \cdot \hat{l})\, d\hat{l}. \tag{20}$$

We can drop all dependencies except the one on the normal $\hat{n}$ and we get the following two dimensional environment map:

$$L_{\text{diffuse}}(\underline{x}; \hat{n}) = \frac{k_d}{\pi} \int_\Omega L_{in}(\hat{l})(\hat{n} \cdot \hat{l})\, d\hat{l}. \tag{21}$$

This environment map accurately stores the diffuse illumination at the point $\underline{x}$. It is only two-dimensional and it is indexed by the surface normal. An example is depicted in Figure 15.

Figure 15: *Bust with diffuse environment map.*

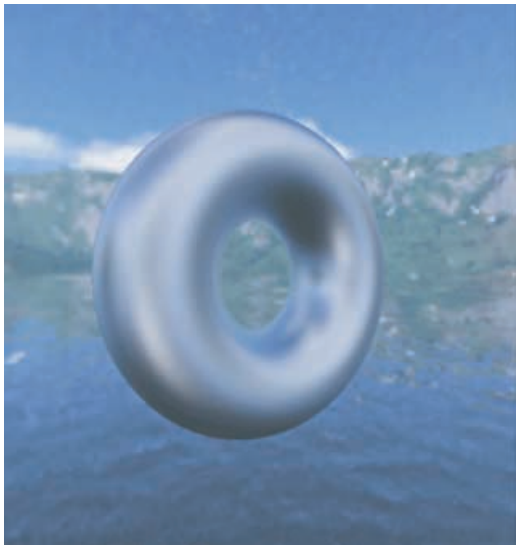

Figure 16: *Torus with a Phong (N = 30) reflection.*

7.3.3. Phong Environment Maps

Heidrich [35] and Miller [62] used the original (specular) Phong reflection model [73] to prefilter environment maps. The specular Phong BRDF is given by:

$$f_r(\hat{v}, \hat{l}) := k_s \frac{(\vec{r}_v(\hat{n}) \cdot \hat{l})^N}{(\hat{n} \cdot \hat{l})}, \tag{22}$$

where $\hat{r}_v(\hat{n})$ is the reflected viewing-direction in world-space. The parameters k_s and N are used to control the shape and size of the lobe. Using the Phong model, Equation 13 becomes

$$L_{\text{phong}}(\underline{x}; \hat{v}, \hat{n}, \hat{t}) = \int_\Omega k_s \frac{(\hat{r}_v(\hat{n}) \cdot \hat{l})^N}{(\hat{n} \cdot \hat{l})} L_{in}(\hat{l})(\hat{n} \cdot \hat{l})\, d\hat{l} \tag{23}$$

$$= k_s \int_\Omega (\hat{r}_v(\hat{n}) \cdot \hat{l})^N L_{in}(\hat{l})\, d\hat{l}. \tag{24}$$

Using the same reparameterization as with the specular BRDF, we get:

$$L_{\text{phong}}(\underline{x}; \hat{r}_v) = k_s \int_\Omega (\vec{r}_v \cdot \hat{l})^N L_{in}(\hat{l})\, d\hat{l}. \tag{25}$$

Although the Phong model is not physically based, the reflections make a surface look metallic, although at glancing angles one expects sharper reflections than provided by this model. This indexing via the reflection vector $\hat{r}_v$ is also used for specular environment maps and is therefore supported in OpenGL [68] via the spherical, parabolic and cube map parameterizations. An example is shown in Figure 16.

Miller et al. [62] and Heidrich et al. [35] proposed a weighted sum of a diffuse and a Phong environment map to get a complete illumination model. They also propose to add a Fresnel term so that the ratio between the diffuse and glossy reflections can vary with different viewing angles:

$$L_o(\hat{r}_v, \hat{n}) = (1 - F(\hat{r}_v \cdot \hat{n}))L_{\text{diffuse}} + F(\hat{r}_v \cdot \hat{n})L_{\text{phong}} \tag{26}$$

In this way, a wider range of materials can be created.

7.3.4. Environment Maps with Approximations of Isotropic BRDFs

Kautz and McCool [42] extended the Phong environment maps idea to other isotropic BRDFs by approximating them with a special class of BRDFs:

$$f_r(\hat{v}, \hat{l}) := p(\hat{n} \cdot \hat{r}_v(\hat{n}), \hat{r}_v(\hat{n}) \cdot \hat{l}), \tag{27}$$

where p is an approximation to a given isotropic BRDF, which is not only isotropic, but also radially symmetric about $\hat{r}_v(\hat{n})$, and therefore only depends on two parameters.

Now consider Equation 13 using this reflectance function:

$$L_{\text{iso}}(\underline{x}; \hat{v}, \hat{n}, \hat{t}) = \int_\Omega p(\hat{n} \cdot \hat{r}_v(\hat{n}), \hat{r}_v(\hat{n}) \cdot \hat{l}) L_{in}(\hat{l})(\hat{n} \cdot \hat{l})\, d\hat{l}. \tag{28}$$

The authors then make the assumption that the used BRDF is fairly specular, i.e. the BRDF is almost zero everywhere, except when $\hat{r}_v(\hat{n}) \approx \hat{l}$. Using this assumption they reason that $\hat{n} \cdot \hat{r}_v(\hat{n}) \approx \hat{n} \cdot \hat{l}$. Now the equation can be reparameterized and rewritten the following way:

$$L_{\text{iso}}(\underline{x}; \hat{r}_v, \hat{n} \cdot \hat{r}_v) = (\hat{n} \cdot \hat{r}_v) \int_\Omega p(\hat{n} \cdot \hat{r}_v, \hat{r}_v \cdot \hat{l}) L_{in}(\hat{l})\, d\hat{l}, \tag{29}$$

which is only three dimensional. Additionally, they proposed the following approximation to a given isotropic BRDF:

$$f_r(\hat{v}, \hat{l}) := F(\hat{n} \cdot \hat{r}_v(\hat{n}))p(\hat{r}_v(\hat{n}) \cdot \hat{l}). \tag{30}$$

This approximates a BRDF with a constant lobe (defined by p) that is scaled by a factor which depends on the angle between $\hat{n}$ and $\hat{r}_v(\hat{n})$. An environment map prefiltered with this model is only two dimensional:

$$L_{\text{iso}}(\underline{x}; \hat{r}_v, \hat{n} \cdot \hat{r}_v) = (\hat{n} \cdot \hat{r}_v)F(\hat{n} \cdot \hat{r}_v) \int_\Omega p(\hat{r}_v \cdot \hat{l})L_{in}(\hat{l})\, d\hat{l}. \tag{31}$$

It is two dimensional because the dependence on $(\hat{n} \cdot \hat{r}_v)$ can be moved outside the integral. It is sufficient to multiply the two additional factors onto the prefiltered environment map during rendering, which is in fact a generalization to the Fresnel term used earlier.

Figure 17 shows a copper teapot. There is almost no visual difference between the ray traced and 3D approximation. The simpler 2D approximation does show some minor differences.

This technique has the big advantage that it can use approximations of arbitrary isotropic BRDFs and achieves interactive frame rates. Off-specular peaks can also be incorporated into this technique; $\hat{r}_v$ is then substituted with a function $\hat{o}(\hat{r}_v)$ that changes the lookup direction accordingly. An additional Fresnel factor like Miller [62] and Heidrich [35] proposed is not needed because real physically based BRDFs can be used. The 2D approximation is directly equivalent to Phong prefiltered environment maps with a separate Fresnel factor, but a more generally shaped lobe is used and the Fresnel factor is directly computed from real BRDF data. In contrast, the 3D approximation does not compute a separate Fresnel factor, instead it is incorporated into the approximation, which allows the shape of the lobe to vary.

Depending on the BRDF, the quality of the approximation varies. For higher quality approximations Kautz and McCool also propose a multilobe approximation, which basically results in several prefiltered environment maps which have to be summed.

For instance, if a BRDF is to be used, that is based on several separate surface phenomena (e.g. has retro-reflections, diffuse reflections, and glossy reflections) each part has to be approximated separately, since no radially symmetric approximation can be found for the whole BRDF. This again corresponds to the technique by Miller or Heidrich, just that it is based on a real BRDF, see Equation 26.

7.3.5. Tabulated Environment Maps with Isotropic BRDFs

Cabral et al. [11] use a similar technique, called *reflection-space image-based rendering*, which also assumes an isotropic BRDF. They prefilter an environment map for different fixed viewing directions, resulting in several view-dependent environment maps. In contrast to the previous approach, they actually use a four dimensional environment map:

$$L_{\text{tab}}(\underline{x}; \hat{v}, \hat{n}) = \int_\Omega f_r(\hat{\omega}(\hat{v}, \hat{n}, \hat{t}^*), \hat{\omega}(\hat{l}, \hat{n}, \hat{t}^*)) \cdot L_{in}(\hat{l})(\hat{n} \cdot \hat{l})\, d\hat{l}, \tag{32}$$

where $\hat{t}^*$ is an arbitrarily chosen tangent, which is possible since the BRDF is assumed to be isotropic. This four

Figure 17: *Copper teapot reflecting environment. Top: Computed using ray tracing. Middle: 2D approximation. Bottom: 3D Approximation.*

Figure 18: *Rendering result with a tabulated environment map. Courtesy Marc Olano.*

dimensional environment map is sparsely sampled in $\hat{v}$. A two dimensional spherical map is extracted from this four dimensional map for every new viewpoint. This map corresponds to one specific viewing direction and is generated using warping. This new view-dependent environment map is then applied to an object. The warping compensates for the undersampled viewing directions and aligns discontinuities, so no visible artifacts occur.

Using a sparse four dimensional environment map makes it unnecessary to approximate the factor $(\hat{n} \cdot \hat{l})$. The necessary warping requires high-end graphics hardware to achieve interactive frame rates, but the final rendering can be done with standard OpenGL environment mapping, which is why an intermediate two dimensional spherical map is generated.

Warping is based on the (most likely) central reflection direction of the BRDF (the reflected viewing direction and the surface normal are mentioned). This assumption fails for BRDFs that have off-specular reflections.

As mentioned before, the generated two dimensional environment map is view-dependent, so theoretically the reflective object needs to be viewed with an orthographic projection or otherwise the reflections are incorrect, since the reflection directions are computed based on an infinite viewer.

Latta and Kolb [51] use the same input data, but compress the 4D environment map using a homomorphic factorization [59]. Rendering is very simple and works in real-time, as it only requires blending of various textures. Of course this technique potentially introduces compression artifacts, although they seem to be negligible.

7.3.6. Environment Maps with Anisotropic Banks BRDF

The first technique for rendering anisotropic reflections from environment maps was proposed by Kautz et al. [46].

Figure 19: *Anisotropic teapot.*

They chose to use the Banks model [5], which is simple and depends only on dot products. It yields a three dimensional environment map if self shadowing is excluded. The specular part of the BRDF is given by:

$$f_r(\vec{v}, \vec{l}) := (\sqrt{1 - (\vec{l} \cdot \vec{t})^2}\sqrt{1 - (\vec{v} \cdot \vec{t})^2} - (\vec{l} \cdot \vec{t})(\vec{v} \cdot \vec{t}))^N, \tag{33}$$

where N controls the sharpness of the highlight. Inserting this into the environment map equation (13) yields:

$$L_{\text{banks}}(\mathbf{x}; \vec{v}, \vec{n}, \vec{t}) = \int_\Omega (\sqrt{1 - (\vec{l} \cdot \vec{t})^2}\sqrt{1 - (\vec{v} \cdot \vec{t})^2} - (\vec{l} \cdot \vec{t})(\vec{v} \cdot \vec{t}))^N L_i(\mathbf{x}; \vec{l})(\vec{n}, \vec{l})\, d\vec{l}. \tag{34}$$

To decrease the dimensionality of this environment map, the authors discard the self-shadowing term $< \vec{n}, \vec{l} >$, and then reparameterization gives the following three dimensional environment map:

$$L_{\text{banks}}(\mathbf{x}; \vec{t}, (\vec{v} \cdot \vec{t})) = \int_\Omega (\sqrt{1 - (\vec{l} \cdot \vec{t})^2}\sqrt{1 - (\vec{v} \cdot \vec{t})^2} - (\vec{l} \cdot \vec{t})(\vec{v} \cdot \vec{t}))^N L_i(\mathbf{x}; \vec{l})\, d\vec{l}. \tag{35}$$

Now we have an anisotropic prefiltered environment map. This anisotropic environment map can then be rendered at interactive rates if the hardware supports three dimensional texturing. In Figure 19 a teapot is shown, which was rendered with an anisotropic prefiltered environment.

Since the self-shadowing term is omitted, an object using this environment map does reflect light from behind it. This is usually not noticeable unless a bright light source shines "through" the object.

7.3.7. Environment Maps with Anisotropic Lafortune BRDF

Recently, McAllister et al. [58] have proposed an environment mapping technique based on the Lafortune model [49]. As we will see, it is a special case of the Kautz and McCool [42] environment mapping method, see Equation 31. The specular part of the Lafortune BRDF can be written as follows:

$$f_r(\hat{v}_\omega, \hat{l}_\omega) := k_s\, p(\hat{o}(\hat{v}_\omega) \cdot \hat{l}_\omega), \tag{36}$$

$$p(\hat{r}_\omega \cdot \hat{l}_\omega) := (\hat{r}_\omega \cdot \hat{l}_\omega)^N, \tag{37}$$

$$\hat{r}_\omega = \hat{o}(\hat{v}_\omega) := \begin{pmatrix} C_x \hat{v}_{\omega,x} \\ C_y \hat{v}_{\omega,y} \\ C_z \hat{v}_{\omega,z} \end{pmatrix}. \tag{38}$$

Substituting this into Equation 13 and moving $(\hat{n} \cdot \hat{r}_v(\hat{n}))$ outside the integral (making the same approximation as Kautz and McCool [42], see above) yields:

$$L_{\text{laf}}(\underline{x}; \hat{v}, \hat{n}, \hat{t}) = (\hat{n} \cdot \hat{r}_v(\hat{n}))k_s \int_\Omega p(\hat{o}(\hat{v}_\omega) \cdot \hat{l}) L_{in}(\hat{l})\, d\hat{l}, \tag{39}$$

and then reparameterizing in terms of $\hat{r}_\omega$ this gives us:

$$L_{\text{laf}}(\underline{x}; \hat{r}_\omega, \hat{n} \cdot \hat{r}_v) = (\hat{n} \cdot \hat{r}_v)k_s \int_\Omega p(\hat{r}_\omega \cdot \hat{l}) L_{in}(\hat{l})\, d\hat{l}. \tag{40}$$

This is only a 2D environment map, since the integral can be precomputed for all directions $\hat{r}_\omega$ (the factor $(\hat{n} \cdot \hat{r}_v)$ need not to be included in the precomputation). It has to be accessed with $\hat{r}_\omega$, which basically is the reflected viewing direction, but potentially including off-specular peaks (similar to $o(\hat{r}_{\hat{v}})$ from Section 7.3.4).

McAllister et al. noticed that using a normalized function $\hat{o}()$ makes the precomputed environment maps completely independent of the parameters $C_{x,y,z}$; only k_s needs to be scaled to take the norm of $\hat{o}()$ into account. Hence an environment map only needs to be prefiltered for different N, whereas $C_{x,y,z}$ are moved outside the integral and can be accounted for during lookup.

This technique and the method by Kautz and McCool are similar, but they use different methods to find the functions $p()$ and $o()$. Kautz and McCool use a numerical approximation, whereas McAllister et al. note that for the Lafortune model, these functions can be derived analytically.

7.4. Environment Mapping for Dynamic Lighting

For interactive applications it would be interesting if environment map filtering could be done on the fly using graphics hardware. This means that if the scene changes, glossy reflections would change accordingly. In this survey, we will only deal with the accelerated filtering of a *given* environment map. It is obvious [80] that environment maps can be generated on the fly. Live video capturing of an environment map is also conceivable. For example the Omnicam [67] directly captures an environment as parabolic map.

In the following we will describe different methods that allow glossy reflections from dynamic lighting through fast filtering of environment maps.

7.4.1. Diffuse Environment Maps

Equation 21 describes how an original environment map L_{in} has to be filtered with a diffuse BRDF. Basically, a hemispherical cosine kernel has to be applied to L_{in}. Since this is a low-frequency kernel, the resulting filtered environment map will also be low-frequency (see Figure 15).

Ramamoorthi and Hanrahan [76] proposed to do this filtering in frequency space in order to exploit the low-frequency nature of the kernel. This can be done using the spherical harmonics [22] (SH) basis $\{y_i(\hat{l})\}$, which is the analogue on the sphere to the Fourier basis on the line or circle.

An environment map is represented in the spherical harmonics basis as:

$$L_{in}(\hat{l}) = \sum_i L_i\, y_i(\hat{l}), \tag{41}$$

where the coefficients L_i are computed numerically:

$$L_i = \int_\Omega L_{in}(\hat{l}) y_i(\hat{l})\, d\hat{l}. \tag{42}$$

As it turns out [76], convolving an environment map in spherical harmonics with the hemispherical cosine kernel is very simple. In fact, convolution and lookup with the surface normal $\hat{n}$ can be combined:

$$L_{\text{diffuse}}(\hat{n}) \approx \frac{k_d}{\pi} \sum_{i=0}^{9} \hat{A}_i L_i\, y_i(\hat{n}), \tag{43}$$

where $\hat{A}_0 = \pi$, $\hat{A}_{1;2;3} = \frac{2}{3}\pi$, and $\hat{A}_{4;5;6;7;8} = \frac{1}{4}\pi$. For a complete derivation, please see Ramamoorthi and Hanrahan [76].

This filtering and lookup step is so simple that it can be easily implemented in a vertex shader on modern graphics hardware. Per-pixel evaluation is also possible but not necessary, since $L_{\text{diffuse}}(\hat{n})$ only varies slowly.

7.4.2. Environment Maps with Approximations of Isotropic BRDFs

Kautz et al. [46] noticed that the Phong model, as well as the isotropic approximations of BRDFs [42], presented in Equation 31, correspond to a circular and radially symmetric

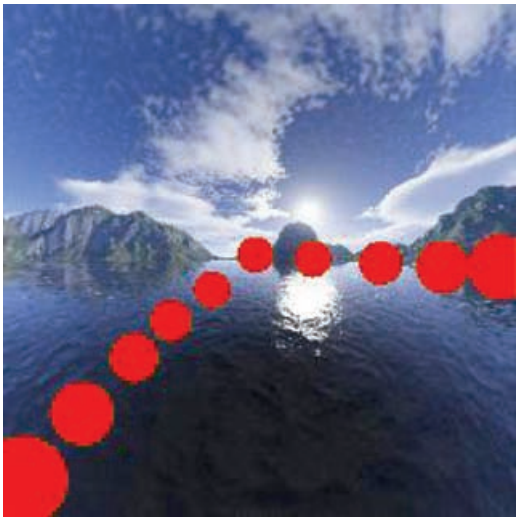

Figure 20: *Variation of originally shift-invariant filter kernel when mapped to a parabolic map (only one hemisphere is shown).*

filter kernel over the sphere. I.e., the filter kernels are shift-invariant over the sphere.

The OpenGL imaging subset supports shift-invariant two dimensional filters of certain sizes [68]. Unfortunately, this feature of graphics hardware cannot be directly applied to speed up the filtering process defined in Equation 31. The imaging subset performs the shift-invariant convolution on a flat 2D texture, whereas environment map filtering is defined over the sphere.

Hence, for hardware accelerated filtering we have to choose an environment map representation that keeps the filter shift-invariant in the 2D texture domain. It turns out that the dual paraboloid mapping proposed by Heidrich and Seidel [34] comes close to this desired property. An infinitesimal circular filter kernel which is mapped from the parabolic environment map back to the hemisphere is again circular, i.e. it is a conformal mapping. A (negligible) distortion occurs depending on the radius and the position of the filter.

Although the shape of the filter almost remains the same in the parabolic space, the radius of the filter kernel varies with the distance d (measured from the center of the map, see Figure 20). The ratio between the smallest filter radius and largest filter radius is about 2. The authors overcome this problem by generating two prefiltered environment maps, one with the smallest (yields map S) and one with the largest necessary filter size (yields map L). Then they blend between both prefiltered environment maps. The value with which we need to blend between both maps is different for different pixels in the parabolic environment map, but it depends only on the distance d and is always d^2. For a pixel in the center of the paraboloid this means that we use 0% of map L and a 100% of map S; for a pixel with distance $d = 0.5$ to the center of the parabolic map, we use 25% of map L and 75% of map S, and so on.

This algorithm maps easily onto graphics hardware; see [46] for more detail.

Figure 16 was done with this technique. The authors report rendering speeds of 70 fps for the Phong model with an exponent of $N = 10$ and 11 fps for an exponent $N = 500$ on an NVIDIA GeForce 4. For smaller exponents (corresponding to larger filter kernels), the authors use a downsampled version of the parabolic environment map, which explains the difference in speed.

7.4.3. Frequency-Space Environment Maps with Isotropic BRDFs

Cabral's method [11] in Section 7.3.5 is based on the observation that the variation of the presented 4D environment is much smaller in $\hat{v}$ than the variation in $\hat{n}$. Ramamoorthi and Hanrahan [77] make the additional observation, that if the (isotropic) BRDF and also the environment map is reparameterized by the reflected viewing direction $\hat{r}_{\hat{v}}$, this variation becomes even smaller:

$$L_{\text{fs}}(\underline{x}; \hat{r}_{\hat{v}}, \hat{v}) = \int_{\Omega} f_r(\hat{\omega}(\hat{v}, \hat{n}^+, \hat{t}^*), \hat{\omega}(\hat{l}, \hat{n}^+, \hat{t}^*)) \cdot L_{in}(\hat{l})(\hat{n}^+ \cdot \hat{l})\, d\hat{l}, \tag{44}$$

$$\text{with } \hat{n}^+ = \frac{\hat{v} + \hat{r}_{\hat{v}}}{||\hat{v} + \hat{r}_{\hat{v}}||}, \tag{45}$$

then the variation in $\hat{v}$ is even smaller. As we have seen for Phong BRDFs, there is no variation at all, it is constant for all $\hat{v}$ reducing the environment map to 2D.

Ramamoorthi and Hanrahan decided to use spherical harmonics [22] to represent the variation in $\hat{v}$ and tabulate the environment map for all directions $\hat{r}_{\hat{v}}$, i.e. the directions corresponding to the reparameterized surface normals. They store the spherical harmonics coefficients of the spherical function, representing exit radiance:

$$L_{\text{fs}}(\underline{x}; \hat{r}_{\hat{v}}, \hat{v}) \approx \sum_i B_i(\hat{r}_{\hat{v}}) y_i(\hat{v}). \tag{46}$$

They call this representation a spherical harmonic reflection map (SHRM). They report that a quadratic or cubic expansion is sufficient for many BRDFs to achieve more than 90% accuracy.

If a distant viewer is assumed, i.e. $\hat{v}$ is constant for every point on the object, then this sum can be computed interactively (in software) resulting in a standard environment map parameterized by $\hat{r}_{\hat{v}}$, but which is only valid for the current view direction $\hat{v}$. The same assumption is also made in Cabral et al.'s work [11].

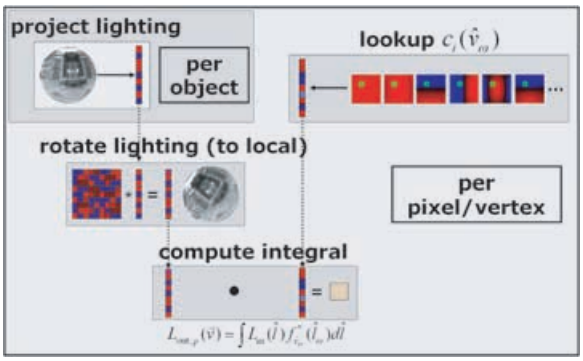

Figure 21: ***Overview.*** *The lighting coefficients are rotated to the local tangent frame at every point on the object, BRDF coefficients are read from texture maps for the local viewing direction, and then an inner product between the coefficient vectors is computed resulting in exit radiance.*

Ramamoorthi and Hanrahan report that converting an environment map into a SHRM takes between .1 seconds and up to 2 seconds, depending on the BRDF. This means that dynamic lighting can be used for some (low-frequency) BRDFs.

7.4.4. Frequency-Space Environment Maps with Anisotropic BRDFs

Another technique based on spherical harmonics was presented by Kautz et al. [45]. We start again with the general filtered environment maps from Section 7.2 and derive this method:

$$L_{\text{sh}}(\hat{v}) = \int_\Omega L_{in}(\hat{l}) f_r(\hat{v}_\omega, \hat{l}_\omega) \max(0, \hat{n} \cdot \hat{l})\, d\hat{l} \qquad (47)$$

$$= \int_\Omega L_{in}(\hat{l}) f^*(\hat{v}_\omega, \hat{l}_\omega)\, d\hat{l}, \qquad (48)$$

where f^* is the BRDF product, i.e. the BRDF times the dot product between $\hat{n}$ and $\hat{l}$.

First parameterize f^* by the local viewing direction $\hat{v}_\omega$ to get spherical functions, which are represented in the SH basis via

$$f^*_{\hat{v}_\omega}(\hat{l}_\omega) \approx \sum_{i=1}^{n^2} c_i(\hat{v}_\omega) y_i(\hat{l}_\omega), \qquad (49)$$

where $y_i(\hat{l}_\omega)$ are again the SH basis functions. The SH coefficients c_i are tabulated in terms of the local view direction $\hat{v}_\omega$. For each $\hat{v}_\omega$, the result is a vector of n^2 ($n = 5$ in their case) coefficients representing a spherical function, as in the upper right of Figure 21. The vectors are stored in (multiple) 2D textures in terms of $\hat{v}_\omega$ (parabolic map).

Figure 22: *Brushed metal head (Ashikhmin-Shirley BRDF* [3]*) in various lighting environments.*

If the incident illumination L_{in} is also represented as a SH coefficient vector $\vec{L}$, the lighting integral (Equation 48) becomes a simple dot-product [83] between the coefficient vectors.

Inconveniently, f^* is represented using the local surface tangent frame which varies over the object while the incident lighting typically uses a global coordinate system shared across the whole object. Thus to perform the integration, we first need to rotate the lighting into this local frame at each p. Fortunately, this can be performed with a matrix-vector multiplication [45]

$$L^\omega_{p,i} = \sum_{j=1}^{n^2} \mathcal{R}_{p,ij} L_j, \qquad (50)$$

where $\mathcal{R}_p$ is an $n^2 \times n^2$ rotation matrix [97]. Then the lighting integral reduces to:

$$L_{\text{sh}}(\hat{v}_\omega) = \sum_{i=1}^{n^2} L^\omega_{p,i} c_i(\hat{v}_\omega). \qquad (51)$$

Figure 21 shows the whole rendering pipeline for this technique. On older GPUs the rotation into the local coordinate system is only possible on the CPU due to instruction count limits in the vertex shader. The final dot product can be performed in a pixel or vertex shader. Results of this technique are depicted in Figure 22. Rendering times vary between 6 fps and 120 fps, depending on the implementation (fixed lighting or fixed view is considerably faster as the rotation into the local frame can be precomputed).

In contrast to the previous method, this method has the advantage that arbitrary anisotropic BRDFs can be used. The drawback of this method is that only low-frequency incident illumination can be used if interactive rates are to be achieved. Otherwise the coefficient vectors and rotation matrices become prohibitively large. Another advantage is that spatially varying BRDFs can be incorporated, by simply tabulating the coefficient vectors of different BRDFs (i.e., the c_i) in texture maps.

Lehtinen and Kautz [52] speed up the computation by expressing the BRDF product function as a matrix as well. This is achieved by doubly-projecting the BRDF into SH [92]. This BRDF matrix can then be combined with the rotation matrix $\mathcal{R}$. Furthermore they perform a change of basis from SH to a locally supported basis. This finally yields the *matrix radiance transfer* approach:

$$L_{\text{mrt}}(\hat{v}_\omega) = \sum_{i=1}^{M} g_j(\hat{v}_\omega)(\mathcal{A}_p \cdot \vec{L})_j, \tag{52}$$

where g_j is the new locally supported basis over the sphere and the matrix $\mathcal{A}_p$ is the combined rotation, BRDF, and change of basis matrix. Since the basis is locally supported, not all M basis functions contribute to the exit radiance. Only the contributing basis functions need to be taken into account (in their case, always only four). By additionally compressing the matrices $\mathcal{A}_p$ using PCA, they achieve speed ups of about a factor of 10.

PCA was also used by Sloan et al. [84] to compress and speed up rendering. This technique performs the PCA only on a local neighborhood to enable better compression (the local neighborhood has very likely the same shading characteristics). They also show evaluation of their compressed data on graphics hardware.

Unfortunately, even with PCA, both these methods still require a lot of memory.

7.5. Transfer (Shadowing, Interreflections)

In this section, we will explain how shadows and interreflections can be incorporated into global incident illumination.

7.5.1. Precomputed Radiance Transfer

Here we present the technique of Sloan et al. [83], which builds on the techniques from the previous sections.

Sloan et al. assume that the object is illuminated by distant low-frequency illumination $L_s(\hat{l})$, represented for example by an environment map. To compute exit radiance from a point p on an object's surface, the integral

$$\begin{aligned} L_{out}(\hat{v}) &= \int_\Omega L_s(\hat{l}) V_p(\hat{l}) f_r(\hat{v}, \hat{l}) \max(0, \hat{n} \cdot \hat{l})\, d\hat{l} \\ &= \int_\Omega L_s(\hat{l}) V_p(\hat{l}) f_r^*(\hat{v}, \hat{l})\, d\hat{l} \end{aligned} \tag{53}$$

needs to be evaluated at each p. All terms and variables have been explained before, except $V_p(\hat{l})$, which is the visibility function — zero for directions along which the environment cannot be seen due to self-shadowing and one if the environment can be seen.

In the general case, this integral is expensive to compute, since the visibility function $V_p(\hat{l})$ changes at every point on the object. Fortunately, under the assumption that the object is rigid, this function remains constant for each point, and thus has to be computed only once. Furthermore, if the incident lighting is represented as a coefficient vector in the spherical harmonics basis, the *transferred radiance* $L_p^*(\hat{l}) = L_{in}(\hat{l}) V_p(\hat{l})$, or more precisely its coefficient vector in the SH basis, can be computed with a matrix-vector multiplication from the SH projection coefficients $\vec{L}$ of the incident lighting. This *transfer matrix* $\mathcal{T}_p$, which varies with the surface location p, can also include interreflection effects in addition to self-shadowing when determining transferred radiance [83]. Transferred radiance is then computed by:

$$L_p^*(\hat{l}) \approx \sum_i (\mathcal{T}_p \vec{L}) y_i(\hat{l}).$$

The transferred radiance is usually required to be represented in the local tangent space of a vertex (see Section 7.4.4); this is again achieved by multiplication with the high-dimensional rotation matrix $\mathcal{R}_p$. We denote the transferred radiance SH vector in local coordinates by $\vec{L}_p^{*\omega}$:

$$\vec{L}_p^{*\omega} = \mathcal{R}_p \mathcal{T}_p \vec{L}.$$

All that remains to be done is the integration against the BRDF. This is e.g. possible with the technique proposed by Kautz et al. [45] (Section 7.4.4):

$$L_{\text{prt}}(\hat{v}_\omega) = \sum_{i=1}^{n^2} L_{p,i}^{*\omega} c_i(\hat{v}_\omega). \tag{54}$$

The only difference to before, is the inclusion of the additional transfer matrix. The remaining computations are the same as in the work of Kautz et al. [45], see Equation 51. For an example of self-shadowing from environment maps, please see Figure 23. Alternatively, the matrix radiance transfer technique can be used to compute final exit radiance (incorporating the transfer matrix) [52].

Figure 23: *Bust with self-shadowing. Compare to Figure 15.*

Sloan et al. [83] proposed a similar solution for the Phong BRDF, based on convolution (performed in the frequency domain). However, since it is only valid for the physically incorrect Phong model, we omit the details here.

Sloan et al. [83] also propose a special solution for diffuse surfaces. In this case, the BRDF is simply a constant and self-shadowing boils down to a dot-product between the SH incident lighting coefficients and the SH coefficients of $k_d/\pi V_p(\hat{l})(\hat{n} \cdot \hat{l})$.

Ng et al. [69] use the same idea as Sloan et al. for diffuse surfaces, except they project into a Haar wavelet basis. For about 100 coefficients, they achieve very sharp shadows, which the SH basis is not capable of (at least not with a reasonable number of coefficients). On the other hand, the SH basis is better suited if only few coefficients are used, because SH only blurs the shadows, which is less disturbing than "quantized" shadows which you would get from too few Haar wavelet coefficients.

7.6. Other Techniques

A few other techniques have been proposed for interactive rendering of glossy reflections which are not based on environment maps. Diefenbach and Badler [21] used multi-pass methods (Monte Carlo integration) to generate glossy reflections. Photon maps [38] have been used by Stürzlinger and Bastos [87]; photons were "splatted" and weighted with an arbitrary BRDF. Precomputed glossy reflections were stored in surface light fields by different authors [63,95,13]. Bastos et al. [6] used a convolution filter in screen-space to produce glossy reflections. Lischinski and Rappoport [55] used a large collection of low resolution layered depth images to store view-dependent illumination.

7.7. Comparison

In this section, we classify all real-time shading algorithms for global illumination. The categories for the classification are listed in Section 6.5. The actual classification is listed in Figure 11. All the algorithms with a '*g*' are for global illumination and were described in this section.

Again, it is hard to decide which algorithm is the overall best. It again depends on the application. For diffuse surfaces, the algorithm by Ramamoorthi et al. [76] is very good if shadowing does not need to be included. If diffuse surfaces with self-shadowing are desired, then Ng et al.'s [69] or Sloan et al.'s [83] algorithm is very well-suited.

If arbitrary BRDFs are to be included (with or without shadowing), then one of the SH precomputed radiance transfer methods [52,83,84] is preferred if you can live with the restriction to low-frequency SH (differences are only in speed and difficulty of implementation).

Higher-frequency (dynamic) illumination is possible with Kautz et al.'s [46] method, at the cost of only working with approximations of isotropic BRDFs and no self-shadowing.

If the lighting is static, then the method by Latta and Kolb [51] is best. It allows arbitrary isotropic BRDFs to be rendered in real-time but with good quality.

8. Conclusions

In this STAR, we have presented all recently proposed algorithms for real-time shading. These algorithms either allow local illumination (point light sources) or global illumination (usually assumed to be distant). We have classified the algorithms in order to show what the advantages and disadvantages of each algorithm are. There is no clear "winner". It depends on the desired effects and the desired rendering speed. For further reading, the reader is referred to the following books: Olano et al. [72], Akenine-Möller and Haines [2], and Fernando and Kilgard [23].

There is still a lot of work to be done in this area. The goal of real-time rendering globally illuminated virtual worlds has not been reached yet. The presented algorithms show the way to go: precompute parts of the necessary computation to save processing power later. Unfortunately, this is very difficult for completely dynamic worlds.

References

1. T. Akenine-Möller and U. Assarsson. Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges. In *13th Eurographics Workshop on Rendering*, pages 309–318, June 2002.

2. T. Akenine-Möller and E. Haines. *Real-Time Rendering*. AK Peters Ltd, Natick, MA, 2002.

3. M. Ashikhmin and P. Shirley. An Anisotropic Phong BRDF Model. *Journal of Graphics Tools*, 5(2):25–32, 2000.

4. U. Assarsson and T. Akenine-Möller. A Geometry-based Soft Shadow Volume Algorithm using Graphics Hardware. *ACM Transactions on Graphics (Proceedings SIGGRAPH)*, 22(3):511–520, July 2003.

5. D. Banks. Illumination in Diverse Codimensions. In *Proceedings SIGGRAPH*, pages 327–334, July 1994.

6. R. Bastos, K. Hoff, W. Wynn, and A. Lastra. Increased Photorealism for Interactive Architectural Walkthroughs. In J. Hodgins and J. Foley, editors, *1999 ACM Symposium on Interactive 3D Graphics*, pages 183–190. ACM SIGGRAPH, April 1999.

7. P. Beckmann and A. Spizzichino. *The Scattering of Electromagnetic Waves from Rough Surfaces*. McMillan, 1963.

8. J. Blinn. Simulation of Wrinkled Surfaces. In *Proceedings SIGGRAPH*, pages 286–292, August 1978.

9. J. Blinn. Models of Light Reflection For Computer Synthesized Pictures. In *Proceedings SIGGRAPH*, pages 192–198, July 1977.

10. J. Blinn and M. Newell. Texture and Reflection in Computer Generated Images. *Communications of the ACM*, 19:542–546, 1976.

11. B. Cabral, M. Olano, and P. Nemec. Reflection Space Image Based Rendering. In *Proceedings SIGGRAPH*, pages 165–170, Los Angeles, California, August 1999.

12. N. Carr, J. Hall, and J. Hart. GPU Algorithms for Radiosity and Subsurface Scattering. In *Graphics Hardware 2003*, pages 51–59, July 2003.

13. W.-C. Chen, J.-Y. Bouguet, M. H. Chu, and R. Grzeszczuk. Light Field Mapping: Efficient Representation and Hardware Rendering of Surface Light Fields. In *Proceedings SIGGRAPH*, pages 447–456, July 2002.

14. P. Christensen. Note #35: Ambient occlusion, image-based illumination, and global illumination. In *Photo-Realistic RenderMan Application Notes*, 2002.

15. M. Cohen and J. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press Professional, Cambridge, MA, 1993.

16. R. Cook and K. Torrance. A Reflectance Model for Computer Graphics. In *Proceedings SIGGRAPH*, pages 307–316, August 1981.

17. R. L. Cook, T. Porter, and L. Carpenter. Distributed Ray Tracing. In *Proceedings SIGGRAPH*, pages 137–145, Minneapolis, Minnesota, July 1984.

18. F. Crow. Shadow Algorithms for Computer Graphics. In *Proceedings SIGGRAPH*, pages 242–248, July 1977.

19. K. Dana, B. van Ginneken, S. Nayar, and J. Koenderink. Reflectance and Texture of Real-World Surfaces. *ACM Transactions on Graphics*, 18(1):1–34, January 1999.

20. K. Daubert, W. Heidrich, J. Kautz, J.-M. Dischler, and H.-P. Seidel. Efficient Light Transport Using Precomputed Visibility. *IEEE Computer Graphics and Applications*, 23(3):28–37, May 2003.

21. P. Diefenbach and N. Badler. Multi-Pass Pipeline Rendering: Realism For Dynamic Environments. In M. Cohen and D. Zeltzer, editors, *1997 ACM Symposium on Interactive 3D Graphics*, pages 59–70. ACM SIGGRAPH, April 1997.

22. A. Edmonds. *Angular Momentum in Quantum Mechanics*. Princeton University, Princeton, NJ, 1960.

23. R. Fernando and M. Kilgard. *The Cg Tutorial*. Addison-Wesley, Boston, MA, 2003.

24. J. Foley, A. van Damme, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Inc., 2nd edition, 1990.

25. A. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann, 1995.

26. N. Greene. Applications of World Projections. In *Proceedings Graphics Interface*, pages 108–114, May 1986.

27. N. Greene. Environment Mapping and Other Applications of World Projections. *IEEE Computer Graphics & Applications*, 6(11):21–29, November 1986.

28. P. Haeberli and M. Segal. Texture Mapping As A Fundamental Drawing Primitive. In *Fourth Eurographics Workshop on Rendering*, pages 259–266. Eurographics, June 1993.

29. Xuejun Hao, Thomas Baby, and Amitabh Varshney. Interactive Subsurface Scattering for Translucent Meshes. In *Proceedings 2003 ACM Symposium on Interactive 3D Graphics*, page to appear, april 2003.

30. W. Heidrich, S. Brabec, and H.-P. Seidel. Soft Shadow Maps for Linear Lights. In *11th Eurographics Workshop on Rendering*, pages 269–280, June 2000.

31. W. Heidrich, K. Daubert, J. Kautz, and H.-P. Seidel. Illuminating Micro Geometry Based on Precomputed Visibility. In *Proceedings SIGGRAPH*, July 2000.

32. W. Heidrich, H. Lensch, and H.-P. Seidel. Light Field-based Reflections and Refractions. In *Tenth Eurographics Rendering Workshop*, June 1999.

33. W. Heidrich and H.-P. Seidel. Efficient Rendering of Anisotropic Surfaces Using Computer Graphics Hardware. In *Image and Multi-dimensional Digital Signal Processing Workshop (IMDSP) '98*, 1998.

34. W. Heidrich and H.-P. Seidel. View-Independent Environment Maps. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 39–45, 1998.

35. W. Heidrich and H.-P. Seidel. Realistic, Hardware-accelerated Shading and Lighting. In *Proceedings SIGGRAPH*, pages 171–178, August 1999.

36. W. Heidrich, R. Westermann, H.-P. Seidel, and T. Ertl. Applications of Pixel Textures in Visualization and

Realistic Image Synthesis. In *Symposium on Interactive 3D Graphics*, 1999.

37. D. Immel, M. Cohen, and D. Greenberg. A Radiosity Method for Non-Diffuse Environments. In *Proceedings SIGGRAPH*, pages 133–142, 1986.

38. H. W. Jensen. Global Illumination using Photon Maps. In *Seventh Eurographics Rendering Workshop 1996*, pages 21–30, June 1996.

39. H. W. Jensen, S. Marschner, M. Levoy, and P. Hanrahan. A Practical Model for Subsurface Light Transport. In *Proceedings SIGGRAPH*, pages 511–518, August 2001.

40. J. Kajiya. The Rendering Equation. In *Proceedings SIGGRAPH*, pages 143–150, August 1986.

41. J. Kautz and M. McCool. Interactive Rendering with Arbitrary BRDFs using Separable Approximations. In *Tenth Eurographics Workshop on Rendering*, pages 281–292, June 1999.

42. J. Kautz and M. McCool. Approximation of Glossy Reflection with Prefiltered Environment Maps. In *Proceedings Graphics Interface*, pages 119–126, May 2000.

43. J. Kautz and M. McCool. Interactive Rendering with Arbitrary BRDFs using Separable Approximations. In *10th Eurographics Rendering Workshop 1999*, pages 281–292, June 1999.

44. J. Kautz and H.-P. Seidel. Towards Interactive Bump Mapping with Anisotropic Shift-Variant BRD Fs. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware 2000*, pages 51–58, August 2000.

45. J. Kautz, P.-P. Sloan, and J. Snyder. Fast, Arbitrary BRDF Shading for Low-Frequency Lighting Using Spherical Harmonics. In *13th Eurographics Workshop on Rendering*, pages 301–308, June 2002.

46. J. Kautz, P.-P. Vázquez, W. Heidrich, and H.-P. Seidel. A Unified Approach to Prefiltered Environment Maps. In *Eleventh Eurographics Workshop on Rendering*, pages 185–196, June 2000.

47. J. Kautz, C. Wynn, J. Blow, C. Blasband, A. Ahmad, and M. McCool. Achieving Real-Time Realistic Reflectance. *Game Developer Magazine*, January 2001.

48. M. Kilgard. *A Practical and Robust Bump-mapping Technique for Today's GPUs*. NVIDIA Corporation, April 2000. Available from `http://www.nvidia.com`.

49. E. Lafortune, S.-C. Foo, K. Torrance, and D. Greenberg. Non-Linear Approximation of Reflectance Functions. In *Proceedings SIGGRAPH*, pages 117–126, August 1997.

50. E. Lafortune and Y. Willems. Bidirectional Path Tracing. In *Compugraphics*, pages 95–104, 1993.

51. L. Latta and A. Kolb. Homomorphic Factorization of BRDF-based Lighting Computation. In *Proceedings SIGGRAPH*, pages 509–516, July 2002.

52. Jaakko Lehtinen and Jan Kautz. Matrix Radiance Transfer. In *Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 59–64, 2003.

53. H. Lensch, M. Goesele, P. Bekaert, J. Kautz, M. Magnor, J. Lang, and H.-P. Seidel. Interactive Rendering of Translucent Objects. In *Proceedings of Pacific Graphics 2002*, pages 214–224. IEEE Computer Society, 2002.

54. E. Lindholm, M. Kilgard, and H. Moreton. A User-Programmable Vertex Engine. In *Proceedings SIGGRAPH*, pages 149–158, August 2001.

55. D. Lischinski and A. Rappoport. Image-Based Rendering for Non-Diffuse Synthetic Scenes. In *Nineth Eurographics Workshop on Rendering*, pages 301–314. Eurographics, June 1998.

56. T. Malzbender, D. Gelb, and H. Wolters. Polynomial texture maps. In *Proceedings SIGGRAPH*, pages 519–528, August 2001.

57. B. Mark. Hardware Shading Language Course: NVIDIA Programmable Graphics Technology. In *SIGGRAPH Course Notes*, July 2002.

58. D. McAllister, A. Lastra, and W. Heidrich. Efficient Rendering of Spatial Bi-directional Reflectance Distribution Functions. In *Proceedings Graphics Hardware*, pages 79–88, September 2002.

59. M. McCool, J. Ang, and A. Ahmad. Homomorphic Factorization of BRDFs for High-Performance Rendering. In *Proceedings SIGGRAPH*, pages 171–178, August 2001.

60. Tom Mertens, Jan Kautz, Philippe Bekaert, Hans-Peter Seidel, and Frank Van Reeth. Interactive Rendering of Translucent Deformable Objects. In *Eurographics Symposium on Rendering*, June 2003.

61. Microsoft Corporation. *DirectX 9.0 SDK*, November 2002. Available from `http://www.microsoft.com/directx`.

62. G. Miller and R. Hoffman. Illumination and Reflection Maps: Simulated Objects in Simulated and Real

Environments. In *SIGGRAPH Course Notes – Advanced Computer Graphics Animation*, July 1984.

63. G. Miller, S. Rubin, and D. Ponceleon. Lazy Decompression of Surface Light Fields for Precomputed Global Illumination. In *Nineth Eurographics Workshop on Rendering*, pages 281–292. Eurographics, June 1998.

64. M. Minnaert. Photometry of the Moon. In *Planets and Satellites*, pages 231–248. University Chicago Press, 1961.

65. J. Mitchell. *Radeon 9700 Shading*. ATI Technologies Inc., July 2002. Available from `http://www.ati.com`.

66. G. Müller, J. Meseth, and R. Klein. Compression and Real-Time Rendering of Measured BTFs Using Local PCA. In *Proceedings of Vision, Modeling and Visualisation 2003*, November 2003.

67. S. Nayar. Catadioptric Omnidirectional Camera. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, june 1997.

68. J. Neider, T. Davis, and M. Woo. *OpenGL - Programming Guide*. Addison-Wesley, 1993.

69. Ren Ng, Ravi Ramamoorthi, and Pat Hanrahan. All-Frequency Shadows Using Non-linear Wavelet Lighting Approximation. *ACM Transactions on Graphics*, page to appear, July 2003.

70. NVIDIA Corporation. *NVIDIA OpenGL Extension Specifications*, April 2003. Available from `http://www.nvidia.com`.

71. NVIDIA Corporation. *NVIDIA SDK*, May 2003. Available from `http://www.nvidia.com`.

72. M. Olano, J. Hart, W. Heidrich, and M. McCool. *Real-Time Shading*. AK Peters Ltd, Natick, MA, 2002.

73. B.-T. Phong. Illumination for Computer Generated Pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

74. T. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon Mapping on Programmable Graphics Hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.

75. Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002.

76. R. Ramamoorthi and P. Hanrahan. An Efficient Representation for Irradiance Environment Maps. In *Proceedings SIGGRAPH*, pages 497–500, August 2001.

77. R. Ramamoorthi and P. Hanrahan. Frequency Space Environment Map Rendering. In *Proceedings SIGGRAPH*, pages 517–526, July 2002.

78. M. Sattler, R. Sarlette, and R. Klein. Efficient and Realistic Visualization of Cloth. In *Eurographics Symposium on Rendering 2003*, pages 167–178, June 2003.

79. M. Segal and K. Akeley. *The OpenGL* Graphics System: *A Specification (Version 1.2.1)*, 1999.

80. SGI. Iris performer. `http://www.sgi.com/software/performer/brew/envmap.html`.

81. P. Shirley, B. Wade, P. Hubbard, D. Zareski, B. Walter, and D. Greenberg. Global Illumination via Density Estimation. In *6th Eurographics Workshop on Rendering*, pages 219–231, June 1995.

82. P.-P. Sloan and M. Cohen. Hardware accelerated horizon mapping. *11th Eurographics Workshop on Rendering*, pages 281–286, June 2000.

83. P.-P. Sloan, J. Kautz, and J. Snyder. Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments. In *Proceedings SIGGRAPH*, pages 527–536, July 2002.

84. Peter-Pike Sloan, Jesse Hall, John Hart, and John Snyder. Clustered Principal Components for Precomputed Radiance Transfer. *ACM Transactions on Graphics*, page to appear, July 2003.

85. B. G. Smith. Geometrical Shadowing of a Random Rough Surface. *IEEE Transactions on Antennas and Propagation*, 15(5):668–671, September 1967.

86. M. Steigleder. Factorization of the Ashikhmin BRDF for Real-Time Rendering. *Journal of Graphics Tools*, 7(4):61–68, 2002.

87. W. Stürzlinger and R. Bastos. Interactive Rendering of Globally Illuminated Glossy Scenes. In *Eighth Eurographics Workshop on Rendering*, pages 93–102. Eurographics, June 1997.

88. F. Suykens, K. vom Berge, Lagae A, and P. Dutré; Interactive Rendering with Bidirectional Texture Functions. *Computer Graphics Forum*, 22(3):463–472, September 2003.

89. I. Wald, T. Kollig, C. Benthin, A. Keller, and P. Slusallek. Interactive global illumination. In *13th Eurographics Workshop on Rendering*, pages 9–20, June 2002.

90. I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. *Computer Graphics Forum*, 20(3):153–164, 2001.

91. G. Ward. Measuring and Modeling Anisotropic Reflection. In *Proceedings SIGGRAPH*, pages 265–272, July 1992.

92. S. Westin, J. Arvo, and K. Torrance. Predicting Reflectance Functions From Complex Surfaces. In *Proceedings SIGGRAPH*, pages 255–264, July 1992.

93. L. Williams. Casting Curved Shadows on Curved Surfaces. In *Proceedings SIGGRAPH*, pages 270–274, August 1978.

94. T.-T. Wong, P.-A. Heng, S.-H. Or, and W.-Y. Ng. Image-based Rendering with Controllable Illumination. In *Eighth Eurographics Workshop on Rendering*, pages 13–22, June 1997.

95. D. Wood, D. Azuma, K. Aldinger, B. Curless, T. Duchamp, D. Salesin, and W. Stuetzle. Surface Light Fields for 3D Photography. In *Proceedings SIGGRAPH*, pages 287–296, July 2000.

96. C. Wynn. BRDF-Based Lighting. Technical report, NVIDIA Corporation, 2000.

97. R. Zare. *Angular Momentum: Understanding Spatial Aspects in Chemistry and Physics*. Wiley, New York, 1987.

98. S. Zhukov, A. Iones, and G. Kronin. An Ambient Light Illumination Model. In *Eurographics Rendering Workshop 1998*, pages 45–56, June 1998.