ADVANCED TOPICS IN SCIENCE AND TECHNOLOGY IN CHINA

Hujun Bao
Wei Hua

Real-Time Graphics Rendering Engine

ZHEJIANG UNIVERSITY PRESS
浙江大学出版社
Springer

国家科学技术学术著作出版基金资助出版

Hujun Bao
Wei Hua

Real-Time Graphics Rendering Engine

With 66 figures, 11 of them in color

ZHEJIANG UNIVERSITY PRESS
浙江大学出版社

Springer

ADVANCED TOPICS
IN SCIENCE AND TECHNOLOGY IN CHINA

ADVANCED TOPICS
IN SCIENCE AND TECHNOLOGY IN CHINA

Zhejiang University is one of the leading universities in China. In Advanced Topics in Science and Technology in China, Zhejiang University Press and Springer jointly publish monographs by Chinese scholars and professors, as well as invited authors and editors from abroad who are outstanding experts and scholars in their fields. This series will be of interest to researchers, lecturers, and graduate students alike.

Advanced Topics in Science and Technology in China aims to present the latest and most cutting-edge theories, techniques, and methodologies in various research areas in China. It covers all disciplines in the fields of natural science and technology, including but not limited to, computer science, materials science, life sciences, engineering, environmental sciences, mathematics, and physics.

Hujun Bao
Wei Hua

Real-Time Graphics Rendering Engine

With 66 figures, 11 of them in color

ZHEJIANG UNIVERSITY PRESS
浙江大学出版社

Springer

Authors
Prof. Hujun Bao
State key Lab of Computer
Aided Design and Computer
Graphics at Zhejiang University,
Hangzhou 310058, China
E-mail: bao@cad.zju.edu.cn

Dr. Wei Hua
State key Lab of Computer
Aided Design and Computer
Graphics at Zhejiang University,
Hangzhou 310058, China
E-mail: huawei@cad.zju.edu.cn

ISSN 1995-6819 e-ISSN 1995-6827
Advanced Topics in Science and Technology in China

ISBN 978-7-308-08133-7
Zhejiang University Press, Hangzhou

ISBN 978-3-642-18341-6 e-ISBN 978-3-642-18342-3
Springer Heidelberg Dordrecht London New York

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

A real-time graphics rendering engine is a middleware, and plays a fundamental role in various real-time or interactive graphics applications, such as video games, scientific computation visualization systems, CAD systems, flight simulation, etc. There are various rendering engines, but in this book we focus on a 3D real-time photorealistic graphics rendering engine, which takes 3D graphics primitives as the input and generates photorealistic images as the output. Here, the phrase "real-time" indicates that the image is generated online and the rate of generation is fast enough for the image sequence to be looked like a smooth animation. For conciseness, we use the rendering engine to represent a 3D real-time photorealistic graphics rendering engine throughout this book.

As a rendering engine is a middleware, users are mainly application developers. For application developers, a rendering engine is a software development kit. More precisely, a rendering engine consists of a set of reusable modules such as static or dynamic link libraries. By using these libraries, developers can concentrate on the application's business logic, not diverting attention to rather complicated graphics rendering issues, like how to handle textures or how to calculate the shadings of objects. In most cases, a professional rendering engine usually does rendering tasks better than the programs written by application developers who are not computer graphics professionals. Meanwhile, adopting a good rendering engine in application development projects can reduce the development period, since lots of complex work is done by the rendering engine and, consequently, development costs and risks are alleviated.

In this book we are going to reveal the modern rendering engine's architecture and the main techniques used in rendering engines. We hope this book can be good guidance for developers who are interested in building their own rendering engines.

The chapters are arranged in the following way. In Chapter 1, we introduce the main parts of a rendering engine and briefly their functionality. In Chapter 2, basic knowledge related to developing real-time rendering is introduced. This covers the rendering pipeline, the visual appearance and shading and lighting models. Chapter 3 is the main part of this book. It unveils the architecture of the rendering engine through analyzing the Visionix system, the rendering engine developed by

the authors' team. Lots of details about implementation are also presented in Chapter 3. In Chapter 4, a distributed parallel rendering system for a multi-screen display, which is based on Visionix, is introduced.

In Chapters 5 and 6, two particular techniques for real-time rendering that could be integrated into rendering engines are presented. Chapter 5 presents an overview of real-time rendering approaches for a large-scale terrain, and a new approach based on the asymptotic fractional Brownian motion. Chapter 6 presents a variation approach to a computer oriented bounding box tree for solid objects, which is helpful in visibility culling and collision detection.

This book is supported by the National Basic Research Program of China, also called "973" program (Grant Nos. 2002CB312100 and 2009CB320800) and the National Natural Science Foundation of China (Grant No. 60773184). Additionally, several contributors have helped the authors to create this book.

Dr. Hongxin Zhang and Dr. Rui Wang from CAD&CG State Key Lab, Zhejiang University, China, have made key contributions to Chapter 2 "Basics of Real-time Rendering". Ying Tang from Zhejiang Technology University, China, has done lots of work on tailoring contents, translating and polishing this book.

Special thanks to Chongshan Sheng from Ranovae Technologies, Hangzhou, China, as one of the main designers, for providing a lot of design documents and implementation details of the Visionix system, which is a collaboration between Ranovae Technologies and the CAD&CG State Key Lab.

Many people who work, or have ever studied, at the CAD&CG State Key Lab, Zhejiang University, provided help and support for this book: Rui Wang, Huaisheng Zhang, Feng Liu, Ruijian Yang, Guang Hu, Fengming He, Wei Zhang, Gaofeng Xu, Ze Liang, Yifei Zhu, Yaqian Wei, En Li, and Zhi He.

Hujun Bao
Wei Hua
Hangzhou, China
October, 2010

Contents

1

Introduction

In this chapter, we are going to introduce the main parts of most rendering engines and, briefly, their functionality. Fig. 1.1 shows a classical structure of a rendering engine. In this graph, the rendering engine is composed of offline toolkits and a runtime support environment. The offline toolkit mainly comprises the tools that export data from the third party modeling software and the tools which perform some pre-operations to the Scene Model. These pre-operations include simplification, visibility pre-computation, lighting pre-computing and data compression. Besides these tools, the more advanced rendering engine includes some special effects generators. The main parts supported in Runtime are Scene Model, Scene Model Management, Scene Graph Traversal and Render. The applications call Scene Model Management and Scene Graph Traversal to run the rendering engine. In the following paragraphs, we will give a brief description of the core parts of the runtime support environment.

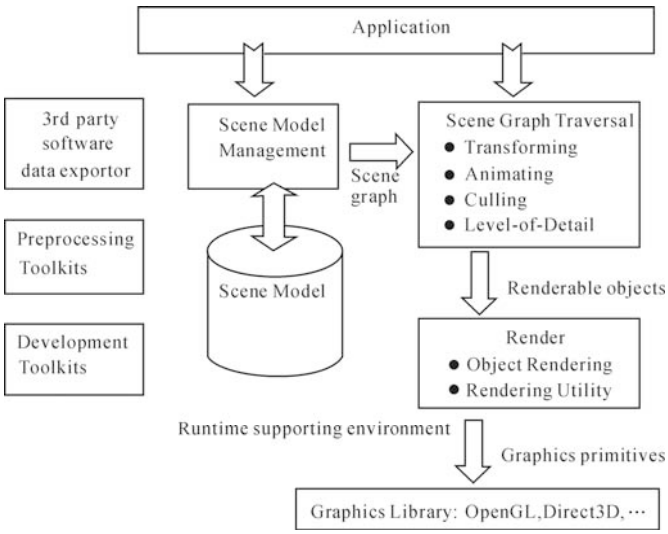

Fig. 1.1 The classical structure of a rendering engine

1.1 Scene Graph Management

Inside a rendering engine, the scene model is the digital depiction of the virtual world in cyberspace. For most rendering engines, the scene model adopts graph data structures, which is called the scene graph. The scene graph is a directed acyclic graph, where nodes represent the entities of the virtual world and arcs represent relations between these entities.

In a scene graph, different kinds of nodes represent different classes of entities. The two most fundamental nodes are renderable objects and light sources. The renderable objects represent the objects that can be displayed on the images produced by rendering engines. The light sources stand for the sources of light, which describe light intensity, emission fashion, position, direction, etc. Given a scene graph, the prime function of rendering engines is to use the light sources to illuminate the renderable objects, and render the renderable objects according to certain viewing parameters.

Undoubtedly, renderable objects are the most important class of entities. In object-oriented architecture, the renderable object is a subclass of the base class entity. To represent various perceptible entities in a virtual world optimally, there is a variety of subclasses of renderable objects. Although these subclasses of renderable objects may look quite different, most of them have two parts in common, geometry and appearance. The geometric part describes the outline, contour, surface or volume of a renderable object. The number of geometric types that can be supported is regarded as an index to measure the performance of a rendering engine. The polygonal mesh or, more precisely, the triangular mesh is the most widely supported geometric representation for all rendering engines with a simple structure. The polygonal mesh can be used to represent most geometric entities and can be easily mapped to graphics hardware. Some more advanced rendering engines adopt a spline surface as the geometric representation to describe finer surfaces. The rendering engine aiming at scientific visualization would support a volumetric dataset. The appearance part describes the optical characteristics of material that constitutes the surface or volume of a renderable object. Many visual effects of renderable objects are dependent on it. Since textures are well supported, on all modern 3D graphics cards the appearance part often uses multiple textures to record various optical properties on surfaces.

The arcs in a scene graph represent the relations between renderable objects. Most rendering engines implement the scene graph by tree structures. In a tree, different types of nodes represent different node relations. The most common relationship is a grouping relationship and the corresponding node is a group node. A group node represents a group of entities and the entities inside the group become the child node of this group node. A grouping relationship is very useful, for it is used to model the hierarchy of the virtual world. In some rendering engines, a group node has a transformation field, which depicts a coordinate transformation for all children in the group's coordinate frame.

Besides a grouping relationship, there is another kind of important relationship

between nodes—reference. The reference here is similar to the reference in C++. The goal of adopting a reference here is to improve the reuse efficiency and decrease the storage size. For example, in order to represent a district with 100 similar houses, a straightforward method is that we first build one mesh to represent one house and then build the other 99 houses by replicating the mesh 99 times with spatial transformations. This method is very simple. However, it consumes a great amount of memory by replicating the mesh multiple times. To solve this problem, most rendering engines adopt a reference, where we first build a mesh for a house, then build 100 nodes. Each node includes a reference to the mesh and the related spatial transformation. In this way, we only need to store one mesh and use references to realize reuse of the meshes multiple times. The references in different engines are realized in different ways, which can be achieved by ID, address or handles.

In order to build the spatial relations of the nodes in a scene graph, we need to build the spatial index to a scene graph. The most often used spatial indices are BSP tree, quad tree, octree and kd tree. With a spatial index we can quickly determine the spatial relations between entity and ray / line, entity and entity, entity and view frustum, including intersection, disjoint or enclosed. With such accelerations, we can obviously improve the time efficiency of scene graph operations, such as object selection and collision detection.

Most rendering engines provide one module, a scene graph manager (different rendering engines may have different names), to help manipulate scene graphs, so as to create, modify, reference, duplicate, rename, search, delete scene graph nodes. The functions of a scene graph manager can be roughly classified into the following categories:

(1) Node lifetime management. This is mainly for creating, duplicating and deleting nodes.

(2) Node field management. This is to provide get / set functions of nodes' fields. Furthermore, it provides more complex field updating functions, such as changing the node's name, which requires solving the name conflicts problems. Applying a transformation such as translation, rotation, scale, or their combinations to nodes is the basic but important function in this category. For some powerful rendering engines, adding and deleting user-defined fields are supported.

(3) Node relation management. This is mainly for grouping / ungrouping nodes, referencing / dereferencing nodes, etc., and collapsing nodes.

(4) Spatial index management. This is to construct and update the spatial index of a scene. Since there are several kinds of spatial index, such as binary partition tree, kd tree and octree, one rendering engine usually implements one spatial index. A local update of the spatial index for dynamic scenes is very crucial for a large scene, for it will save much computational expense.

(5) Query utility. This is for finding nodes by name, type, bounding volume, intersecting ray or other information.

(6) Loader and serializer. Almost every rendering engine has one module to load the scene graph from files. It checks the syntax of the contents of files (some even check the semantics), creates nodes and assembles them to form a scene

graph in the host memory. Most rendering engines tend to define one intrinsic file format, and provide a plug-in of third-part modeling software, such as 3D Studio MAX, Maya, to export their own file. Some of them provide tools to convert other formats into it. Corresponding to the parser module, a rendering engine has a serialize module, which serializes the whole or part of the scene graph into a stream.

1.2 Scene Graph Traverse

The rendering engine works in a cycling manner. For each cycle the rendering engine sets the camera parameters and traverses the scene graph once, during which time it finishes the rendering for one frame. So we only need to set the camera parameters according to the walkthrough path to realize the walkthrough of a scene.

There is one specific module, which we call a traversal manipulator, responsible for scene graph traversal. The traversal manipulator traverses the scene graph node by node. Among the operations done to the nodes, the animation controllers are the most important. They update the states of animated nodes according to the time.

To represent dynamic objects in a virtual world, such as moving vehicles, light flicker, a running athlete and so on, some rendering engines provide various animation controllers, such as a keyframe animation controller, skeletal animation controller, particles controller, etc. Each animated node could have one or several controllers attached. In the traverse of a scene graph, the attached controllers are visited and have the opportunity to execute some codes to make the state of the object up-to-date.

After the operations on the nodes have been done, the traversal manipulator determines which nodes need to be rendered and puts these nodes in a rendering queue. The two most important decisions to made are as follows:

(1) Visibility Culling. By using visibility culling, potentially visible objects are selected and sent to the primitive render, so as to avoid those definitely invisible objects consuming rendering resources. Therefore, visibility culling is an important rendering acceleration technique and is very effective for in-door scenes.

(2) Level-of-detail selection. Level-of-detail technique uses a basic idea to reduce the rendering computation, so that the objects close to the viewpoint are rendered finely and the objects far away from the viewpoint are rendered coarsely. Powered by level-of-detail techniques, one renderable object usually has many versions, each of which has a different level of detail. During the traverse, for each renderable object with LOD, an object version with a proper level of detail is carefully selected according to the distance and viewing direction from the viewpoint to the object, so that the rendering results of the selected object version look almost the same as that of the original object.

1.3 Rendering Queue

Through the traversal manipulator, the to-be-rendered scene graph nodes are stored in the rendering queue and delivered to the render module. The render arranges the nodes in the rendering queue in a proper order, which may be spatial relations from front to back or from back to front, or material types. Most rendering engines regard the underline graphic rendering pipeline as a finite state machine. They arrange the rendering order according to the goal to reduce the switch times of state machines, which improves the rendering speed with the precondition of rendering accuracy.

1.4 Rending Modle

After rearranging the order, the render calls a proper rendering process according to the node types. Generally speaking, there is at least one rendering process for each renderable object, such as triangle meshes, billboards, curves, indexed face sets, NURBS and text. The rendering engine uses one module to manage these rendering processes, which is called a render. The so-called rendering process is actually a set of algorithms, which break down the rendering for a renderable object to a series of rendering statements supported by a bottom graphics library (like OpenGL or Direct3D). A render is not just a simple combination of a set of rendering processes. It has a basic framework to coordinate, arrange and manage the rendering queue and rendering processes. In addition, it includes a series of public rendering utilities, to reduce the difficulty of developing rendering processes for different nodes. The core parts of the render are:

(1) Texture mapping module. This handles a variety of texture mappings, such as multi-texturing, mipmapping, bump mapping, displacement mapping, volumetric texture mapping, procedural texture mapping, etc. Some of the rendering engines also provide texture compression / decompression, texture packing, texture synthesis and other advanced texture related functions.

(2) Shading module. This calculates reflected or refracted light on the object surface covered by a certain material and lit by various light sources, such as point-like omni-light sources, line-like light sources, spotlight sources, directional light sources, surface-like light sources, environmental light sources, etc. The module supports several kinds of illumination models, such as the Phong model, the Blinn model, the Cook and Torrance model, and so on. Adopting a different illumination model usually requires a different appearance model. At the runtime stage, rendering engines only support local illumination, for global illumination is computationally very expensive to achieve in real-time. To simulate global illumination, lightmapping is used by many rendering engines. However, lightmapping is limited to showing diffuse components in static lighting scenarios. Nowadays, precomputed radiosity transfer techniques provide a new way to show

objects with a complex appearance model in dynamic lighting conditions.

(3) Shadows module. As a shadow is a phenomenon of lighting, shadow computation rigorously should be a feature of a lighting and shading module. Nevertheless, if we consider the light source as a viewpoint, the shaded places can be considered as the places invisible to the light source. Therefore, shadow computation by nature is a visibility determination problem, which is a global problem depending on the spatial relations of the entire scene, including object-object and light-object relations. Due to the complexity of this problem, it becomes a separate module in most rendering engines.

Besides the above important modules, some rendering engines provide a series of assistance modules, such as:

(1) Observer controller: To control the observer's position, direction, field of view, motion speed / angular speed, motion path and the characteristics of image sensors.

(2) Special visual effects: To simulate the effects of ground / water explosion, explosion fragments, flashes from weapon firing, traces of flying missiles, rotation of airscrew, airflows of rotating wings, smoke and flames, etc.

(3) Display environment configuration: To support the display devices of CAVE, Powerwall etc. Users can configure the number of displays, their arrangement styles and the stereo eyes' distance. This also supports non-linear distortion correction, cylinder and planar projection display and the edge blending of multi-displays.

2

Basics of Real-Time Rendering

This chapter is concerned with the basics of real-time rendering, namely the graphics rendering pipeline, graphics representation and illumination model. The terms graphics pipeline or rendering pipeline mostly refer to state-of-the-art methods of rasterization-based rendering, supported by commodity graphics hardware. Its main function is to synthesize or to render 2D raster images with given 3D scene information including scene geometry, virtual camera, lights, material as well as different types of texture, etc.

As depicted in Figs. 2.1 – 2.3, different application systems have different graphics hardware architecture. For example, inside a modern personal computer, a typical graphics system includes a CPU, an advanced graphics card with multiple-core GPU and data communication lines between them. The CPU performs pre-processing for graphics data and is responsible for user interaction. Then the processed graphics data are transferred to the main memory and later transferred to the video memory of the graphics card through the PCI-E Bus. After that, the rendering data are efficiently processed by the GPU through the pipeline stages. Finally, the results are outputted from the graphics card and displayed on the user's screen, while inside a modern game console including SONY Play Station 3 and Microsoft XBOX 360, the Power CPU can directly communicate with the GPU without an additional data bus. Thus the GPU is needed for whatever rendering platform and the graphics pipeline is adopted by the GPU for rendering.

2.1 Rendering Pipeline

Rendering, described as a pipeline, consists of several consequential stages. This implies that the speed of the pipeline is determined by its slowest pipeline stage, no matter how fast the other stages may be. Therefore, it is worth analyzing typical rendering pipelines to know their performance features.

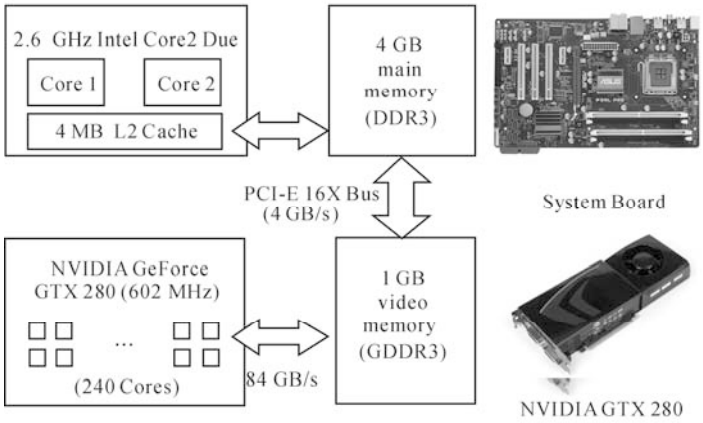

Fig. 2.1 Typical graphics application systems: Modern PC

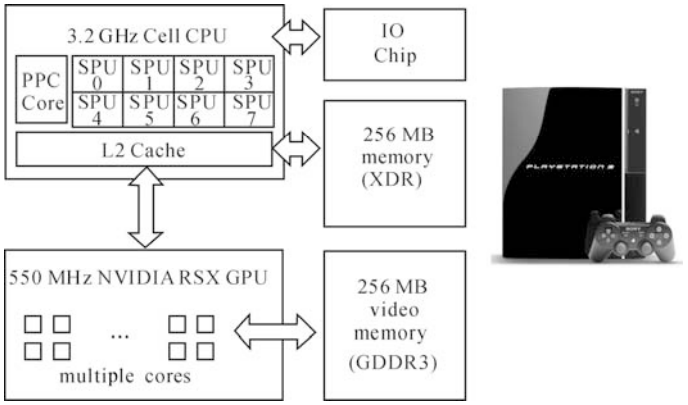

Fig. 2.2 Typical graphics application systems: SONY Play Station 3

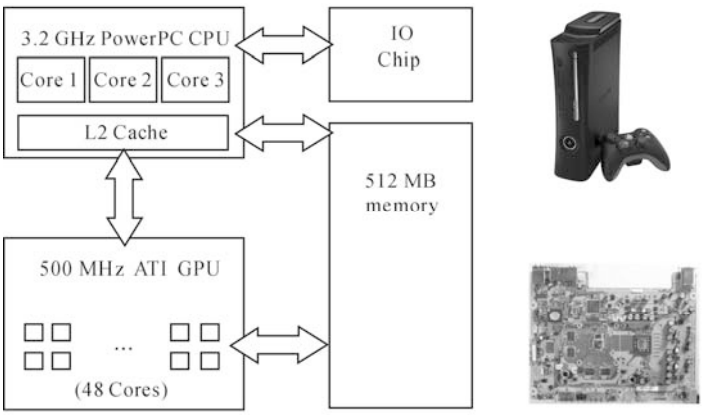

Fig. 2.3 Typical graphics application systems: XBOX 360

When people discuss a rendering pipeline, they may talk about it on two different levels. One is on the software API level. For example, OpenGL API and Direct3D API are two graphics pipeline models accepted as widespread industry standards. Both APIs only provide logical frameworks of how the 3D scene is to be rendered. The other level is the real hardware implementation level, i.e., the actual rendering instructions running on the CPUs and graphics cards. The API level, which defines simple but clear standard rendering models, is easy to understand for the end user. However, as real-time rendering is always a time critical task, a different hardware vendor may provide different solutions and strategies to fully use the horsepower of electronic computing units. This leads to the fact that on the hardware level, the details of a rendering pipeline are always different from what was explained on the API level. Due to this phenomenon, in the following discussions we focus on the logical framework of a rendering pipeline.

2.1.1 Conceptual Rendering Phases

From the viewpoint of graphics architecture, a typical rendering pipeline is divided into three conceptual phases, which are user input, geometry processing and scan conversion phases, and which are presented by Akenine-Möller T *et al*. (Akenine-Möller *et al*., 2008). Moreover, each of these phases consists of several sub-stages. This structure is the core architecture of a real-time rendering engine. Note that, according to the above discussion, in real applications it is unnecessary to have one-to-one correspondence between conceptual rendering stages and functional implementation steps. A typical OpenGL rendering pipeline and corresponding phases are illustrated in Fig. 2.4.

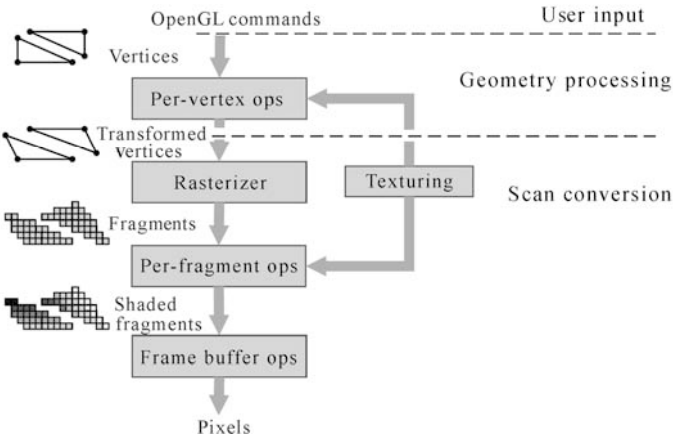

Fig. 2.4 Typical OpenGL rendering pipeline

In the user input phase, the developers are allowed to fully control what the software executes. During this phase, the geometry primitives to be rendered, i.e.,

points, lines and triangles, are fed to the next phase in the rendering pipeline. This is the central task of this phase. The developer can process the data to be fed into the pipeline to improve the rendering performance, like simplifying data or building the hierarchical scene structure. This phase is application-based, where the developer needs to write specific codes to determine the operation while, in the other two phases, the graphic operations are directly implemented on the graphics hardware. With a fixed rendering pipeline, the user cannot control the rendering process in these two stages, providing rendering parameters instead. With a programmable rendering pipeline supported by an advanced GPU, users have more and more flexibility to control the operations inside these two stages, so as to compute the desired effects.

In the geometry processing phase, per-primitive operations, mainly coordinate transformations and per-vertex shading, are performed to map 3D information into 2D screen-based representation. It is worth noting that the geometry phase is a stage with dense computing. With merely a single light source, each vertex may require approximately 100 single float point operations.

In the scan conversion phase, correct color values are computed for each pixel in the raster image with the input of transformed and projected vertices, colors and texture coordinates from the previous phase. During this phase, 2D vertices, lines and polygons are scanned line by line in the screen space with associated depth-values, colors and texture coordinates, and are converted into pixels on the screen. Unlike the previous phase, which handles per-primitive operations, the scan conversion phase handles per-pixel operations. The information for each pixel is stored in the frame buffer, which is a rectangular array of colors (with four components including red, green, blue and alpha). Hence, this process is also known as rasterization. For high-performance graphics, it is critical for the rasterization stage to be implemented in the hardware.

2.1.2 Programmable Rendering Pipeline

Modern graphics hardware provides developers an additional ability to control the rendering pipeline using the power of the graphics processing unit (GPU). Namely, the aforementioned three rendering phases can be customized by the programmable features of a GPU. It is clear that the user input phase is already fully programmable even in a conventional graphics system. Therefore, what GPU techniques tend to do is to provide customizable per-primitive and per-pixel operations in the latter two phases, respectively. This reflects the techniques of vertex shader and fragment (also called pixel) shader.

A vertex shader is a graphics processing function used to customize special effects in a 3D virtual scene by performing mathematical operations on the vertex data of those 3D objects. Each vertex maintains a variety of geometrical and shading information. For example, a vertex is always defined by its location in 3D

coordinates, and may also be defined by colors, surface orientations, textures coordinates and lighting characteristics. Vertex shaders cannot actually change the type of data; they simply change the values of the data, so that a vertex emerges with a different color, different textures, or a different position in space.

Pixel shaders give developers the ability to calculate effects on a per-pixel basis. We can use fragment shaders to create materials and surfaces with high reality. With pixel shaders, the artificial, computerized look of conventional hardware accelerated rendering is replaced by highly realistic surfaces.

To provide more flexible rendering pipelines, geometry shaders are introduced with Shader Model 4.0 of DirectX 10. A geometry shader is a shader program model that can generate new graphics primitives, such as points, lines and triangles, from those primitives sent to the beginning of the rendering pipeline. Now this feature is supported in DirectX 10 and in OpenGL through an extension. Recently, main-stream graphics hardware from nVidia, AMD(ATI) and Intel, such as nVidia GTX 285 and AMD/ATI HD 4870, provides hardware support for geometry shaders.

Geometry shader programs are executed after vertex shaders. They take as input the primitives as a whole, possibly with adjacency information. For example, when operating on triangles, the three vertices are the geometry shader's input. The shader produces zero or more primitives, which are rasterized to fragments. The fragments are ultimately passed to a pixel shader. Typical uses of geometry shaders include point sprite generation, geometry tessellation, shadow volume extrusion and single pass rendering to a cube map.

2.1.3 Geometry Transforms

A geometry transform is a basic tool for manipulating geometry, and plays an important role in real-time rendering. In a virtual world, 3D objects or models are rendered on a 2D screen to be observed by users with a virtual camera. Originally, a model is represented in its own model space. The related coordinates of an object are called model coordinates. This means that the model has not been transformed at all. To be displayed on the screen, a model with its vertices and normals needs to be transformed into a unique global coordinates system or a global space. This space is also called a world space. As there are at least two different types of coordinate systems of one model, model transforms are introduced to unify the representation. After the models have been transformed with their respective model transforms, all models are transformed from the model space to the unique global space. Therefore, all models are represented in the same coordinate system after model transforms.

Virtual cameras are deposited in global space with specific viewing directions. To facilitate projection and clipping, a virtual camera and all the models are transformed with the view transform. In general, after the view transform, the

virtual camera is placed at the origin and aimed at the negative z-axis, with the y-axis pointing upwards and the x-axis pointing to the right. But it is worth noting that a different Application Program Interface (API) may provide different results for the actual position and orientation after the view transform.

The model transform and the view transform are both implemented as 4×4 matrices. And a homogeneous coordinate is used to denote points and vectors for representation consistency and convenience. According to such notation, a vector is represented as $\boldsymbol{v} = (\boldsymbol{v}_x, \boldsymbol{v}_y, \boldsymbol{v}_z, 0)^T$ and a point as $\boldsymbol{p} = (\boldsymbol{p}_x, \boldsymbol{p}_y, \boldsymbol{p}_z, 1)^T$. Therefore, performing a general 3D geometrical transform $\boldsymbol{M}$ on a point $\boldsymbol{p}$ under homogeneous representation can be computed by linear operation as $\boldsymbol{p} = \boldsymbol{M}\boldsymbol{v}$.

For generating complex space transformations, a general solution is to combine several elementary transformations by multiplying the corresponding matrices together. Therefore, in most cases, only several basic transforms need to be considered in a rendering system, which are translation, rotation, scaling, reflection and shearing. The choice of representations is dependent on API support. Moreover, it is worth noting that it is not necessary to represent the transform in matrix form. In fact, rotation matrices can be converted into quaternion representation, which is a powerful tool of rotation representation.

Geometrical transforms require a lot of computation in a graphics pipeline. Therefore, in modern graphics pipelines, most geometric transforms are implemented in graphics hardware.

2.2 Shading

The goal of rendering is to create images that accurately represent the shape and appearance of objects in scenes. Once the geometry and visual information are given, shading is performed for the rendering purpose. Shading is the process of performing lighting computations and determining colors for each pixel. In this section, we present the concepts and definitions required to formulate the problem that the shading algorithm must solve.

2.2.1 Rendering Equation

The distribution of light energy in a scene can be mathematically formulated as the rendering equation. For simplicity, and not involving too many details, we assume that light propagates in the scene instantaneously and there is no participating media, subscattering surface and reradiation material in the scene, hence the exitant light of surface point x is from, and only from, the one shooting at it.

The radiance transported from a surface point $\boldsymbol{x}$ comes from two kinds of sources. One is the radiance emitted by the surface point $\boldsymbol{x}$ and the other is the

radiance that is reflected by the surface at $\boldsymbol{x}$. Let us use the most commonly used formulation of a rendering equation, hemispherical formulation, to explain how light is transported in the scene. Let us assume that $L_e(\boldsymbol{x}, \boldsymbol{\omega}_o)$ represents the radiance emitted in the outgoing direction $\boldsymbol{\omega}_o$, and $L_r(\boldsymbol{x}, \boldsymbol{\omega}_o)$ represents the reflected radiance in that direction $\boldsymbol{\omega}_o$. The reflected radiance $L_r(\boldsymbol{x}, \boldsymbol{\omega}_o)$ is determined by three factors, the incident radiance, the material property of the surface and the normal of point $\boldsymbol{x}$, and expressed as

$$L_r(\boldsymbol{x}, \boldsymbol{\omega}_o) = \int_{\Omega_x} f_x(\boldsymbol{x}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) L(\boldsymbol{\omega}_i) \cos(\boldsymbol{N}_x, \boldsymbol{\omega}_i) \mathrm{d}\boldsymbol{\omega}_i \tag{2.1}$$

where $f_x(\boldsymbol{x}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o)$ is the BRDF function and $L(\boldsymbol{\omega}_i)$ is the incident light along the direction $\boldsymbol{\omega}_i$ and $\cos(\boldsymbol{N}_x, \boldsymbol{\omega}_i)$ is the cosine term of incident light and the normal of point $\boldsymbol{x}$.

The total outgoing radiance at surface point $\boldsymbol{x}$, viewed from a certain outgoing direction, is the integration of the emitted radiance and the radiance reflected at that point in that direction. The outgoing radiance $L_o(\boldsymbol{x}, \boldsymbol{\omega}_o)$ can be represented as follows:

$$L_o(\boldsymbol{x}, \boldsymbol{\omega}_o) = L_e(\boldsymbol{x}, \boldsymbol{\omega}_o) + L_r(\boldsymbol{x}, \boldsymbol{\omega}_o) \tag{2.2}$$

Combining these equations we have

$$L_o(\boldsymbol{x}, \boldsymbol{\omega}_o) = L_e(\boldsymbol{x}, \boldsymbol{\omega}_o) + \int_{\Omega_x} f_x(\boldsymbol{x}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) L(\boldsymbol{\omega}_i) \cos(N_x, \boldsymbol{\omega}_i) \mathrm{d}\boldsymbol{\omega}_i \tag{2.3}$$

Furthermore, we can separate out the direct and indirect illumination terms from the rendering equation. Direct illumination is the illumination directly from the light sources; indirect illumination is the light after bouncing, among in the scene. Hence, the rendering equation is then formulated as:

$$L_r(\boldsymbol{x}, \boldsymbol{\omega}_o) = \int_{\Omega_x} f_x(\boldsymbol{x}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) L(\boldsymbol{\omega}_i) \cos(\boldsymbol{N}_x, \boldsymbol{\omega}_i) \mathrm{d}\boldsymbol{\omega}_i = L_{\text{direct}} + L_{\text{indirect}} \tag{2.4}$$

$$L_{\text{direct}} = \int_{\Omega_x} f_x(\boldsymbol{x}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) L(\boldsymbol{\omega}_i) V(\boldsymbol{\omega}_i) \cos(\boldsymbol{N}_x, \boldsymbol{\omega}_i) \mathrm{d}\boldsymbol{\omega}_i \tag{2.5}$$

$$L_{\text{indirect}} = \int_{\Omega_x} f_x(\boldsymbol{x}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) L(\boldsymbol{\omega}_i) \cos(\boldsymbol{N}_x, \boldsymbol{\omega}_i) \mathrm{d}\boldsymbol{\omega}_i \tag{2.6}$$

where $V(\boldsymbol{\omega}_i)$ is the visibility function specifying the visibility along direction $\boldsymbol{\omega}_i$ and is defined as follows:

$$V(\boldsymbol{\omega}_i) = \begin{cases} 1 & \text{if } \boldsymbol{\omega}_i \text{ is visible,} \\ 0 & \text{if } \boldsymbol{\omega}_i \text{ is NOT visible.} \end{cases} \tag{2.7}$$

Thus, the direct illumination is contributed to by all the emitted radiance visible to the point $\boldsymbol{x}$ along direction $\boldsymbol{\omega}_i$. The indirect illumination is the reflected radiance from all visible points over the hemisphere at point $\boldsymbol{x}$.

2.2.2 Lighting

Light is the product that comes from sources that allow us to see things. In physics, light is electromagnetic radiation but, formally, only visible electromagnetic radiation (wavelength band between 380 nm and 780 nm) is considered as light. Light is variously modeled as quantum, wave or geometric optics. In computer graphics, the most commonly used model of light is the geometric optics model. In this model, the wavelength of light is assumed to be much smaller than the scale of the objects lightened up and the light is treated as traveling along straight paths and traversing instantaneously in the scene.

Besides physical translation, the amount of illumination the light emits, reflects and refracts also needs to be specified. Scientists use radiometry to measure the light. The fundamental radiometric quantity is radiant power, also called flux, and denoted as Φ in watts (W). Such a quantity expresses how much total energy flows from/to/through a surface per unit time. Several concepts are widely used in rendering. Irradiance (E) is the incident radiant power on a surface, per unit surface area (W/m^2). Radiant Exitance or Radiosity is the exitant radiant power per unit surface area and is also expressed in W/m^2. Radiance is flux per unit projected area per unit solid angle, which expresses how much power arrives at (or leaves from) a certain point on a surface, per unit solid angle and per unit projected area.

Our eyes are sensitive to light in the 380 nm and 780 nm wavelength band. However, wavelengths in this visible range are not equally sensed by our eyes. Several scientific underpinnings have been developed to understand the light and the percepived color. Radiometry deals with physical quantities, Photometry explores the sensitivity of human eyes to light and Colorimetry weighs the variations in the spectral distribution of light. The sensitivity of human eyes is given by the so-called luminous efficiency function, which is a bell-shaped function peaking at 550 nm. The convolution of the luminous efficiency function with light is defined as luminance. Luminance is a measure of the light energy, but the eyes are not linearly sensitive to luminance. The perceptional response of a human to light is called brightness. The brightness function of luminance can be roughly depicted by a log or cube root function. Our perception of colors is not equal to all variations in the spectral distribution of light. A commonly used model of our visual system is with three sensors: one signal deals with the quantity of light at shorter wavelengths, one in the mid-range and one at the longer wavelengths. This means that all the colors we perceive can be represented by a vector of three values. It is main because there are three cones in our eyes. The International Commission on Illumination (CIE) developed a set of standard

experiments to measure the three basis functions of light, which are referred to as *X*, *Y* and *Z*. These basis functions are not the sensitivities of the eyes, but any two spectra that map to the same values of *X*, *Y* and *Z* will be sensed as the same color by our eyes. Though the CIE XYS system is useful for a précis description of light, there are limitations in rendering systems, as the *X*, *Y* and *Z* functions are not non-negative. Current color displays, such as color monitors, color projectors etc., work by emitting light from red, green and blue elements. To display a color, the relative intensities of the red, blue and green elements are adjusted so that their sum matches the color to be displayed. The conversion from *XYZ* to RGB light space is linear and can be done with a linear transform. In the rendering system, the RGB light space is the most widely used light system.

The computation of a rendering system requires the specification of three distributions for each light source: spatial, directional and spectral intensity distribution. Directional light is an ideal light source to be positioned infinitely far away from the objects that are being lit. For example, such light can be used to simulate the sun. In contrast, point lights are defined by their position and intensity. According to different values of intensity depending on the directions, point lights are usually categorized as omni-lights that have a constant intensity value or spotlights that have a directional variance of intensity. More realistically, light can be modeled as area lights that are emitted not only from a point but from a region. Recently, the lights captured from the real world are incorporated more and more in the rendering. The environment lighting/map/image is an omnidirectional, high dynamic range image that records the incident illumination conditions at a particular point in space. Such images were used to illuminate synthetic objects with measurements of real light, and later to illuminate real-world people and objects. For more details on capture, evaluation and applications of environment lighting, please refer to (Debevec, 2008). The environment images have since become a widely used tool for realistic lighting in rendering systems.

2.2.3 BRDF

When the three-dimensional models are to be rendered, the models should not only have proper geometrical shapes, but also have desired shading colors. Object materials interact with light in different ways and determines how the incident light is redirected from object's surface. The interaction of light and surfaces is quite complex. People use the bidirectional surface reflectance distribution function (BRDF) (Nicodemus *et al*., 1992) to define the relation between incident and reflected radiance at the same point. The BRDF at a point $\boldsymbol{x}$ is defined as the ratio of the differential outgoing radiance and differential irradiance:

$$f_x(\boldsymbol{x},\boldsymbol{\omega}_i,\boldsymbol{\omega}_o) = \frac{\mathrm{d}L(\boldsymbol{\omega}_o)}{\mathrm{d}E(\boldsymbol{\omega}_i)} = \frac{\mathrm{d}L(\boldsymbol{\omega}_o)}{L(\boldsymbol{\omega}_i)\cos(\boldsymbol{N}_x,\boldsymbol{\omega}_i)\mathrm{d}\boldsymbol{\omega}_i} \tag{2.8}$$

where the terminology is consistent with Eq. (2.1).

Over the past thirty years, many empirical and analytical BRDF models have been developed. Empirical models do not attempt to simulate reflection or scattering in detail from the basic laws of physics. They use a set of mathematical functions that are manipulated by some parameters. Analytical models derive functions with a detailed level of modeling of the surface and physics of light transport rather than try to capture the behavior of a surface as a black box. Here we briefly introduce some BRDF models.

Lambertian Reflectance, or "ideal diffuse", means that an object is totally matte without any shininess and is in a sense the opposite of specular reflection. The diffuse reflectance coefficient is determined by the cosine of the angle between the surface normal and the light direction vector.

Phong Reflectance is presented in a classic paper (Phong, 1975). The model is given in terms of the angle θ_s between the view direction $\boldsymbol{\omega}_o$ and the direction of mirror reflection $\boldsymbol{\omega}_i'$ ($\boldsymbol{\omega}_i'$ is the mirror direction of $\boldsymbol{\omega}_i$ with the normal). It is typically expressed as

$$f_{\text{Phong}}(\boldsymbol{x}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) = \frac{k_d}{\pi} + k_s \cos^p \theta_s \tag{2.9}$$

where k_d is the diffuse coefficient and k_s is the specular coefficient.

Ward Reflectance (Ward, 1992) is similar to the Phong model, except that it uses a power exponential function intead of the cosine term. The exponential term is parameterized by an average slope of the microfacet roughness. It is given by

$$f_{\text{Ward}}(\boldsymbol{x}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) = \frac{k_d}{\pi} + k_s \frac{1}{\sqrt{\cos\theta_i \cos\theta_o}} \frac{\mathrm{e}^{-\tan^2 \theta_h / \alpha^2}}{4\pi\alpha^2} \tag{2.10}$$

where α denotes the standard deviation of the slope, θ_r is the angle between the incident light and the normal and θ_h is the half angle of view direction $\boldsymbol{\omega}_o$ and incident light direction $\boldsymbol{\omega}_i$. A normalized form is further developed by Duer (Duer, 2005) by replacing $\sqrt{\cos\theta_i \cos\theta_o}$ by $\cos\theta_i \cos\theta_o$.

Lafortune Reflectance (Lafortune *et al.*, 1997) is an another generalization of the Phong model that rather than depicting peaks of reflection around the specular direction, it defines lobes around any axis. It can be expressed as

$$f_{\text{Lafortune}}(\boldsymbol{x}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) = \frac{k_d}{\pi} + k_s [C_{xy}(\boldsymbol{u}_x \boldsymbol{v}_x + \boldsymbol{u}_y \boldsymbol{v}_y) + C_z \boldsymbol{u}_z \boldsymbol{v}_z]^p \cdot \frac{n+2}{2\pi[\max(C_{xy}, C_z)]^n} \tag{2.11}$$

where $\boldsymbol{u}$ and $\boldsymbol{v}$ are vectors of the incident light $\boldsymbol{\omega}_i'$ and reflect light $\boldsymbol{\omega}_i'$ represented in the local coordinate where the surface normal is as the z axis, C_{xy} and C_z are coefficients determining the direction and proportions of the lobe, p depicts the

sharpness of the lobes and the last term is added to provide an approximate normalization.

Ashikhmin-Shirley Reflectance (Ashikhmin and Shirley, 2000) extends the Phong model from isotropic to anisotropic and adds an explicit term for Fresnel reflectance that is computed with Schlick's approximation. The model is

$$f_{\text{Ashikhmin-Shirley}}(\boldsymbol{x},\boldsymbol{\omega}_i,\boldsymbol{\omega}_o) = f_s(\boldsymbol{x},\boldsymbol{\omega}_i,\boldsymbol{\omega}_o) + f_d(\boldsymbol{x},\boldsymbol{\omega}_i,\boldsymbol{\omega}_o) \tag{2.12}$$

$$f_s(\boldsymbol{x},\boldsymbol{\omega}_i,\boldsymbol{\omega}_o) = \frac{\sqrt{(p_u+1)+(p_v+1)}}{8\pi} \frac{(\cos\theta_h)^{p_u\cos^2\phi_h + p_v\sin^2\phi_h}}{\cos\theta_h \max(\cos\theta_i,\cos\theta_o)} F(\cos\theta_h) \tag{2.13}$$

$$f_d(\boldsymbol{x},\boldsymbol{\omega}_i,\boldsymbol{\omega}_o) = \frac{28R_d}{23\pi}(1-R_s)(1-(\frac{1-\cos\theta_i}{2})^5)(1-(\frac{1-\cos\theta_i}{2})^5) \tag{2.14}$$

$$F(\cos\theta_h) = R_s + (1-R_s)(1-\cos\theta_h)^5 \tag{2.15}$$

where p_u and p_v are two coefficients to depict the shape of the lobe in the directions of two tangent vectors on the surface that are perpendicular to the surface normal. Rs and Rd are parameters to specify the fractions of incident light reflected specularly and diffusely.

Cook-Torrance Reflectance (Cook and Torrance, 1982) is a BRDF model based on microfacet assumption and geometric optics derivation. The model is given as

$$f_{\text{Cook-Torrance}}(\boldsymbol{x},\boldsymbol{\omega}_i,\boldsymbol{\omega}_o) = \frac{F(\theta_h)D(\theta_h)G(\theta_i,\theta_o)}{\pi\cos\theta_i\cos\theta_o} \tag{2.16}$$

$$D(\theta_h) = \frac{1}{\alpha^2\cos^4\theta_h} \mathrm{e}^{-\tan^2\theta_h/\alpha} \tag{2.17}$$

$$G(\theta_i,\theta_o) = \min(1, \frac{2\cos\theta_h\cos\theta_o}{\cos\theta_{oh}}, \frac{2\cos\theta_h\cos\theta_i}{\cos\theta_{oh}}) \tag{2.18}$$

where F is the Fresnel function, D is the facet distribution model and G is the shadow function, θ_{oh} is the angle between the halfway direction and the direction of the viewer.

2.2.4 Light Transport

Based on the rendering equation, when dealing with simple lighting, such as point light and directional light, the integration can be evaluated efficiently, since only a single direction needs to be sampled. As the lighting becomes more complex, on-the-fly integration becomes intractable. On the other hand, complex lighting such as large area light sources are very important for realistic image synthesis,

because they can produce more natural shadows and shadings than the ideal point or directional light source.

For complex lighting, such as environment lighting, new techniques based on the linearity of the light transport have been developed to accelerate the computation of the rendering equation. The linearity of the light transport says that the output radiance is a linear transformation of the input radiance. More precisely, if we represent the input radiance, or the light source, with a vector $\boldsymbol{a}$, each component encodes the exit radiance in a given position in the scene, targeting a given direction. And similarly, we use a vector $\boldsymbol{L}$ to represent the output radiance, which is the result we are seeking. Then we have:

$$\boldsymbol{L} = \boldsymbol{M}\boldsymbol{a} \tag{2.19}$$

where $\boldsymbol{M}$ is a matrix that transfers the input radiance to the output radiance. The equation can be directly derived from the rendering equation, and $\boldsymbol{M}$ is determined by the geometry and reflectance properties of the objects in the scene. Note that the relationship still holds, even if a more complex type of light transport, such as interaction with participating media, exists in the scene.

Based on the BRDF representation, the radiance transfer $\boldsymbol{M}$ is usually computed into diffuse radiance transfer and glossy radiance transfer separately. If the surface is diffuse, then the exit radiance of x can be represented by a scalar, $(\boldsymbol{L}_o)_x$. A transfer vector $\boldsymbol{M}_x$ can be used to encapsulate the linear transformation on the incident lighting vector, producing the scalar exit radiance via

$$(\boldsymbol{L}_o)_x = \boldsymbol{M}_x\boldsymbol{L} \tag{2.20}$$

$\boldsymbol{M}_x$ is the row of the transfer matrix $\boldsymbol{M}$ that corresponds to $\boldsymbol{x}$, with its component representing the linear influence that a lighting basis function $\boldsymbol{y}_i(\boldsymbol{\omega})$ has on shading at p. Glossy transfer can be defined in a similar way to diffuse transfer. First of all, we need a model for the glossy reflection. The simplest one is a Phong-like model, which models the reflection by a view dependent kernel $\boldsymbol{G}(\boldsymbol{\omega},\boldsymbol{R},r)$. The distribution of the reflected radiance is assumed to be symmetric about the reflection direction, and the parameter r defines the "glossiness" of the specular response. It is more convenient to think of the final shading as a convolution of the transferred incident radiance $\boldsymbol{L}'$ and the kernel $\boldsymbol{G}(\boldsymbol{\omega},(0,0,1),r)$, evaluated at $\boldsymbol{R}$. Note that $\boldsymbol{L}'$ is the projection of the basis function instead of a single scalar in the diffuse case, and the transfer becomes

$$\boldsymbol{L}'_p = \boldsymbol{M}_x\boldsymbol{L} \tag{2.21}$$

$$(\boldsymbol{L}_o)_x = (G(\boldsymbol{s},(0,0,1),r) \otimes \boldsymbol{L}'_x) \cdot \boldsymbol{y}(\boldsymbol{R}) \tag{2.22}$$

$\boldsymbol{M}_x$ is the transfer matrix that transfers the global incident radiance to the local coordinate frame defined at x, with its component $(\boldsymbol{M}_x)_{ij}$ representing the linear influence that a lighting basis function $\boldsymbol{y}_j(\boldsymbol{\omega})$ has on the i-th coefficient of the exit