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

图书在版编目(CIP)数据

实时图形绘制引擎技术 = Real-Time Graphics Rendering Engine : 英文 / 鲍虎军, 华炜著. —杭州 : 浙江大学出版社，2010.12
(中国科技进展丛书)
ISBN 978-7-308-08133-7

Ⅰ. ①实… Ⅱ. ①鲍… ②华… Ⅲ. ①计算机制图－英文 Ⅳ. ①TP391.41

中国版本图书馆CIP数据核字(2010)第227911号

实时图形绘制引擎技术
鲍虎军　华炜　著

责任编辑	黄娟琴
封面设计	俞亚彤
出版发行	浙江大学出版社 网址：http://www.zjupress.com Springer-Verlag GmbH 网址：http://www.springer.com
排　　版	杭州中大图文设计有限公司
印　　刷	浙江印刷集团有限公司
开　　本	710mm×960mm　1/16
印　　张	19.5
字　　数	495
版 印 次	2010 年 12 月第 1 版　2010 年 12 月第 1 次印刷
书　　号	ISBN 978-7-308-08133-7 (浙江大学出版社) ISBN 978-3-642-18341-6 (Springer-Verlag GmbH)
定　　价	130.00 元

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)

Contents

1 Introduction 1

1.1 Scene Graph Management 2
1.2 Scene Graph Traverse 4
1.3 Rendering Queue 5
1.4 Rending Modle 5

2 Basics of Real-Time Rendering 7

2.1 Rendering Pipeline 7
2.1.1 Conceptual Rendering Phases 9
2.1.2 Programmable Rendering Pipeline 10
2.1.3 Geometry Transforms 11
2.2 Shading 12
2.2.1 Rendering Equation 12
2.2.2 Lighting 14
2.2.3 BRDF 15
2.2.4 Light Transport 17
2.3 Summary 19
References 19

3 Architecture of Real-Time Rendering Engine 21

3.1 Overview 21
3.2 Basic Data Type 22
3.2.1 Single-Field Data Type 22
3.2.2 Multiple-Field Data Type 25
3.2.3 Persistent Pointer: TAddress<> 26
3.3 Basics of Scene Model 26
3.4 Entity 28
3.5 Feature 29
3.5.1 IAttributedObject and IFeature 29
3.5.2 IBoundedObject 31

3.5.3 IChildFeature 31
3.5.4 Subclasses of IGroupingFeature 32
3.5.5 Subclasses of IShapeFeature 33
3.5.6 IAnimatedFeature 38
3.5.7 Subclasses of ILightFeature 40
3.5.8 Subclasses of IBindableFeature 40
3.5.9 IGeometryFeature 42
3.5.10 IAppearanceFeature and Related Features 55
3.6 Scene Graph 73
3.7 Spatial Index 75
3.7.1 Relation Schema A 77
3.7.2 Relation Schema B 79
3.8 Scene Model Schema 79
3.9 Scene Model Interface and Implementation 82
3.9.1 Scope of Name and ID 82
3.9.2 Transaction 82
3.9.3 Scene Storage 82
3.9.4 Reference and Garbage Collection 83
3.9.5 Data Visit and Cache 84
3.9.6 Out-of-Core Entity 85
3.9.7 ISceneModel 86
3.9.8 ISceneStorage 89
3.9.9 Implementation of ISceneModel and ISceneStorage 91
3.10 Scene Manipulator 93
3.10.1 Manipulator Functions 94
3.10.2 Usage of Scene Model Manipulator 97
3.11 Traversing Scene Model 98
3.11.1 Traverse via Iterator 98
3.11.2 Traverse via Visitor 107
3.12 Rendering Engine 115
3.12.1 CRenderingEngine 115
3.12.2 The Composition of the CRenderingEngine 119
3.13 Render Queue and Its Manager 122
3.14 Camera Manager 123
3.15 GPU Resources and Its Manipulator 124
3.15.1 Texture Resource 125
3.15.2 Buffer Resource 126
3.15.3 Shader Program 128
3.15.4 GPU Resource Manipulator 128
3.16 Render Target and Its Manager 131
3.17 Render Control Unit 134
3.18 Pre-render and Its Manager 137

3.18.1 IPreRender 137
3.18.2 CPreRenderManager 140
3.19 Render Pipelines and Its Manager 142
3.19.1 IRenderPipeLine 142
3.19.2 Modular Render Pipeline 147
3.19.3 Render Module 157
3.19.4 CRenderPipelineManager 160
3.20 Examples of Pre-render 161
3.20.1 CVFCullingPreRender 161
3.20.2 CMirrorPreRender 163
3.20.3 COoCEntityLoader 165
3.20.4 CFeatureTypeClassifier 169
3.20.5 CRenderQueueElementProcessor 171
3.20.6 CLightCullingPreRender 173
3.21 Examples of Modular Render Pipeline and Render Module 174
3.21.1 CShapeRenderPipeline 175
3.21.2 CShapeRenderModule 176
3.22 Implementation Details of CRenderingEngine 186
3.22.1 Configure 186
3.22.2 Initialize 189
3.22.3 DoRendering 190
3.22.4 OpenSceneModel 190
3.23 Conclusion 191
References 192

4 Rendering System for Multichannel Display 193

4.1 The Overview of Parallel Rendering 193
4.1.1 Client-Server 195
4.1.2 Master-Slave 196
4.2 The Architecture of a Cluster-Based Rendering System 196
4.3 Rendering System Interface 197
4.3.1 vxIRenderingSystem 199
4.3.2 vxIModel 201
4.3.3 vxIUI 218
4.3.4 The Basic Example 231
4.4 Server Manager 233
4.4.1 Functionality 233
4.4.2 Structure 233
4.4.3 CServerManager 236
4.4.4 CServiceRequestManager 236
4.4.5 CServiceRequestTranslator 238
4.4.6 CServiceRequestSender 238

4.4.7 CSystemStateManager, CScreenState and CRenderServerState 240
4.4.8 CServiceRequestSRThreadPool 242
4.4.9 IServiceRequest and Subclasses 243
4.5 Implementation of Rendering System Interface 245
4.5.1 Implementation Principles 245
4.5.2 Example 1: Startup System 246
4.5.3 Example 2: Open Scene Model 247
4.5.4 Example 3: Do Rendering and Swap Buffer 248
4.6 Render Server and Server Interface 250
4.7 Application: the Immersive Presentation System for Urban Planning 251
4.7.1 System Deployment 253
4.7.2 Functionality 254
References 256

5 Optimal Representation and Rendering for Large-Scale Terrain 257

5.1 Overview 258
5.1.1 LOD Model of Terrain 258
5.1.2 Out-of-Core Techniques 262
5.2 Procedural Terrain Rendering 263
5.2.1 An Overview of Asymptotic Fractional Brownian Motion Tree 265
5.2.2 afBm-Tree Construction 268
5.2.3 Procedural Terrain Rendering 270
5.2.4 Application 275
5.3 Conclusion 277
References 277

6 Variational OBB-Tree Approximation for Solid Objects 281

6.1 Related Work 282
6.2 The Approximation Problem of an OBB Tree 283
6.3 Solver for OBB Tree 285
6.3.1 Computation of Outside Volume for Single Bounding Box ... 285
6.3.2 Solver for OBB Tree 287
6.4 Experiments and Results 290
6.5 Conclusion 291
References 292

Index 295

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

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.

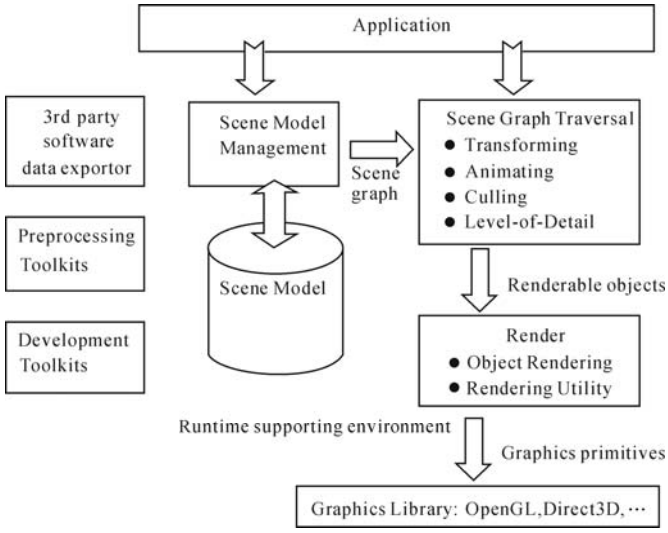

Fig. 1.1 The classical structure of a rendering engine

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.

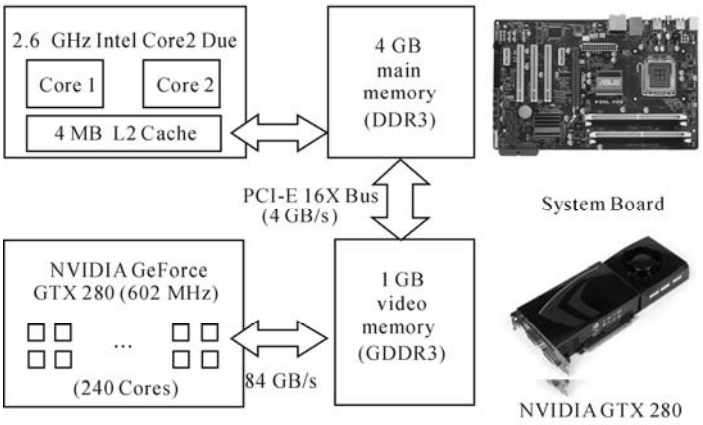

Fig. 2.1 Typical graphics application systems: Modern PC

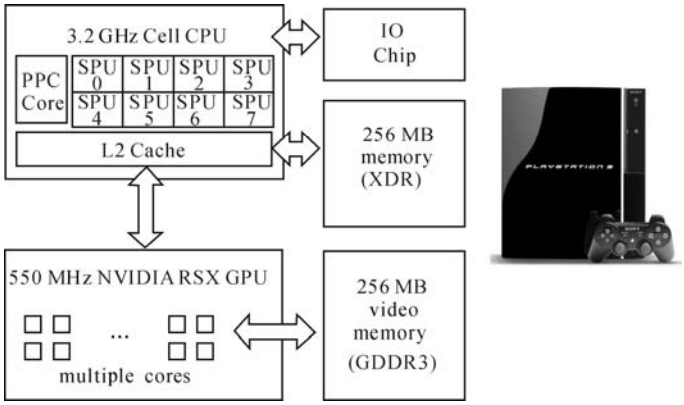

Fig. 2.2 Typical graphics application systems: SONY Play Station 3

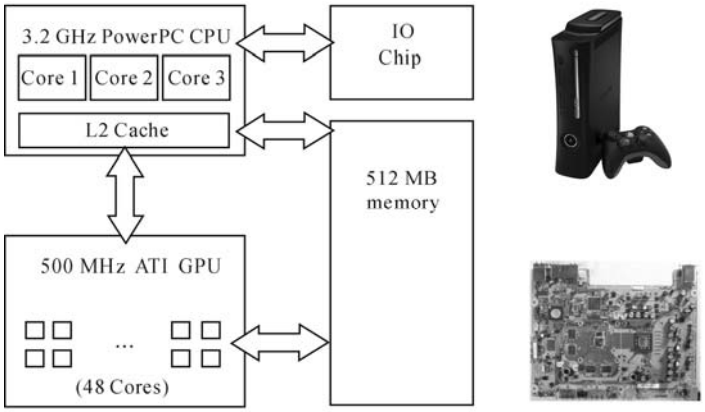

Fig. 2.3 Typical graphics application systems: XBOX 360

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.

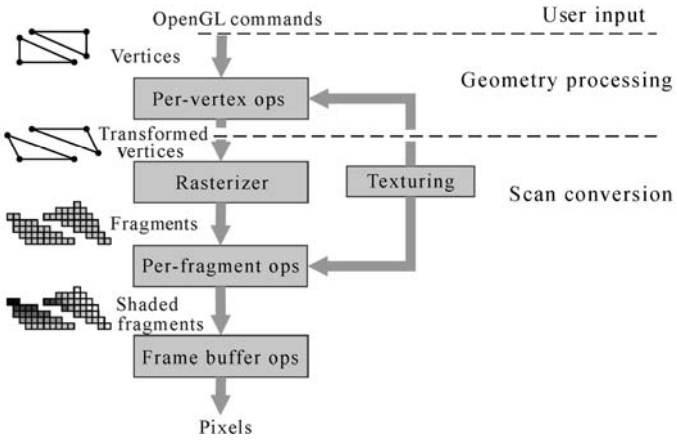

Fig. 2.4 Typical OpenGL rendering pipeline

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

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

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

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

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

lighting vector $\boldsymbol{L}_x'$. For general BRDFs, the reflection can no longer be expressed as a convolution with a kernel that is symmetric about the *z*-axis. The rendering of general BRDFs under environment lighting is more complex and an active research topic (Wang *et al*., 2009).

The linearity of radiance transfer inspired a series of research topics, the precomputed radiance transfer works. The fundamental idea of PRT methods is to tabulate the object's response to light basis functions at a precomputation, and at run-time the integration is reduced to a linear transform of the light basis functions coefficient vector, which exploits the linearity of light transport. Since the computation needed for the shading of each surface element is very simple, graphics hardware can compute the result in a single pass, which then makes real-time rendering feasible. Several light basis functions have been applied, spherical harmonics (Sloan *et al*., 2002), wavelet (Ng *et al*., 2003), spherical radial basis functions (Tsai and Shin, 2006), non-linear spatial basis (Cheslack-Postava *et al*., 2008) etc. For more details of PRT methods and their applications, please refer to (Kautz *et al*., 2005).

2.3 Summary

In this chapter, we introduce several fundamental concepts of real-time rendering. Real-time rendering is always formed as a pipeline composed of consequential data processing and graphics calculations to generate final rasterized images. In order to achieve amazing rendering effects as well as real-time performance, GPU techniques, especially shader techniques, are introduced to utilize programmable pipelines and the parallel computing structure. Virtual geometric 3D models are transformed to be located in a virtual scene by geometrical transformations which are composed by combinations of fundamental transformation matrices. Material and/or other visual properties are then attached to those geometrical models to compute visually pleasant rendering results. As the performance of real-time rendering applications is highly emphasized, tailored shading techniques and BRDF models are employed to achieve real-time rendering results with balanced visual effects.

References

Akenine-Möller T, Haines E, Hoffman N (2008) Real-Time Rendering (3rd Edition). A K Peters, Ltd., Natick

Ashikhmin M, Shirley P (2000) An anisotropic phong BRDF model. J. Graph. Tools 5, 2: 25-32

Cook R, Torrance K (1982) A reflection model for computer graphics. ACM Transactions on Graphics 1: 7-24

Cheslack-postava E, Wang R, Akerlund O, Pellacini F (2008) Fast, realistic lighting and material design using nonlinear cut approximation. ACM Trans. Graph., 27, 5: 1-10

Debevec P (2008) Rendering synthetic objects into real scenes: bridging traditional and image-based graphics with global illumination and high dynamic range photography. In: ACM SIGGRAPH 2008 Classes, New York, 1-10

Duer A (2005) On the Ward model for global illumination. Technical report

Kautz J, Sloan P, Lehtinen J (2005) Precomputed radiance transfer: theory and practice. In: Fujii J, ACM SIGGRAPH 2005 Courses, New York, 1

Lafortune E, Foo SC, Torrance K, Greenberg D (1997) Non-linear approximation of reflectance functions. Computer Graphics, Annual Conference Series, 31: 117-126

Nicodemus FE, Richmond JC, Hsia JJ, Ginsberg IW, Limperis T (1992) Geometrical considerations and nomenclature for reflectance. In: Radiometry, Wolff LB, Shafer SA, Healey G, Eds. Jones And Bartlett Publishers, Inc. Physics-Based Vision: Principles and Practice. Jones and Bartlett Publishers, Sudbury, 94-145

Ng R, Ramamoorthi R, Hanrahan P (2003) All-frequency shadows using non-linear wavelet lighting approximation. In: ACM SIGGRAPH Papers, ACM Press, New York, 376-381

Phong BT (1975) Illumination for computer generated pictures. Commun. ACM 18, 6: 311-317

Sloan P, Kautz J, Snyder J (2002) Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In: Proceedings of the 29th Annual Conference on Computer Graphics and interactive Techniques, San Antonio, Texas

Tsai Y, Shih Z (2006) All-frequency precomputed radiance transfer using spherical radial basis functions and clustered tensor approximation. ACM Trans. Graph. 25, 3: 967-976

Ward G (1992) Measuring and modeling anisotropic reflection. Computer Graphics, Annual Conference Series, 26: 265-273

Wang J, Ren P, Gong M, Snyder J, Guo B (2009) All-frequency rendering of dynamic, spatially-varying reflectance. In: ACM SIGGRAPH Asia 2009 Papers, New York, 1-10

3

Architecture of Real-Time Rendering Engine

3.1 Overview

This chapter presents the architecture of a real-time rendering engine and some implementation details by analyzing the Visionix system, which is the fruit of a long-term project. The Visionix system has been used in several virtual reality applications, such as urban planning, ship steering simulation and interactive games. So far, the Visionix system project is still an active project and is evolving rapidly to keep up with the development of real-time rendering technologies.

We unveil the architecture of a real-time rendering engine based on the analysis of Visionix, which is a good example for readers to learn how to build their own rendering engine. Moreover, we believe the Visionix system has more advantages than many other existing systems.

The most important goal of the Visionix system is to handle large-scale scene models. The major features of the Visionix system are as follows.

(1) It supports plug-in rendering modules and is configurable to meet various application demands.

(2) It has a powerful scene model management subsystem, which is able to manage a large-scale dataset of 3D scenes and is efficient for real-time rendering.

(3) It is an object-oriented system, which provides rich data types to represent various scene objects.

The remaining content of this chapter is divided into two parts. The first part, from Section 3.2 to 3.11, presents important data structures, the scene model representation, and the corresponding storage module. The second part, from Section 3.12 to the end of this chapter, presents the classes which control runtime behaviors of the rendering engine. Both the interfaces and the inner composition of the rendering engine are introduced.

The readers of this chapter should have some knowledge of C++ programming languages, Unified Modeling Language (Pilone and Pitman, 2005), design patterns (Gamma *et al*., 1994), OpenGL (OpenGL Architecture Review Board, 2007; Khronos Group, 2007), C for graphics language (Fernando and Kilgard, 2003) and

basics of computer graphics (Angel, 2005). If the readers have some knowledge of X3D (Brutzman and Daly, 2007; Web3D Consortium, 2008) or VRML (Web3D Consortium, 2003), it would be very helpful for understanding the data structure and the scene graph.

3.2 Basic Data Type

There is a set of basic data types defined in Visionix. These basic data types are fundamental elements for composing any entity class in Visionix. There are two categories, single-field and multi-field data type. All of the basic data types are defined in the name space of BasicType.

3.2.1 Single-Field Data Type

Single-field basic data types are shown in Table 3.1. The prefix SF-means single field, which is the same with X3D's standard naming rules.

Table 3.1 Single-field basic data types

Types	Descriptions
SFBool	A single Boolean value
	typedef SFBool bool;
SFColor, SFColorRGB, SFColorRGBA	The SFColor/SFColorRGB object specifies one RGB (red-green-blue) colour triple. The SFColorRGBA field specifies one RGBA (red-green-blue-alpha) colour quadruple that includes alpha (opacity) information. Each color is in the range 0.0 to 1.0. Alpha values range from 0.0 (fully transparent) to 1.0 (fully opaque)
	struct SFColor { float r, g, b; }; typedef SFColorRGB SFColor; struct SFColorRGBA{ float r, g, b, a; };
SFDouble, SFFloat	The SFDouble and SFFloat variables specify one double-precision floating point number and one single-precision floating point number respectively
	typedef SFDouble double; typedef SFFloat float;
SFChar SFByte SFInt{16|32|64} SFUInt{16|32|64}	The SFChar variable specifies one character, or say 8-bit signed integer. The SFByte specifies one byte, or say 8-bit unsigned integer. The SFInt{16,32,64} specifies one 16-bit, 32-bit or 64-bit signed integer. The SUFInt{16,32,64} specifies one 16-bit, 32-bit or 64-bit unsigned integer

(To be continued)

(Table 3.1)

Types	Descriptions
	typedef SFChar char; typedef SFByte unsigned char; typedef SFInt{16|32|64} __int{16|32|64}; typedef SFUInt{16|32|64} unsigned __int{16|32|64};
SFID	The SFID variable specifies a unique identifier. The Visionix of 64-bit and 32-bit version defines SFID as 64-bit integer and 32-bit integer respectively

	typedef __int32 SFID; // in 32-bit Visionix
	typedef __int64 SFID; // in 64-bit Visionix

SFMatrix3{d|f}, SFMatrix4{d|f}	The SFMatrix3{d|f} object specifies a 3×3 matrix of double-precision or single-precision floating point numbers. The SFMatrix4{d|f} specifies a 4×4 matrix of double-precision or single-precision floating point numbers. These matrices are all organized in column-major fashion
	struct SFMatrix3d{ double m[9]; }; struct SFMatrix3f{ float m[9]; }; struct SFMatrix4d{ double m[16]; }; struct SFMatrix4f{ float m[16]; };
SFRotation3{d|f}	The SFRotation3{d|f} object is a quadruple of double-precision or single-precision floating point numbers, specifying one arbitrary rotation in 3D space. The first three values specify a normalized rotation axis vector about which the rotation takes place. The fourth value specifies the amount of right-handed rotation about that axis in radians. Give a SFRotaiton3d R(x,y,z,a), -R(x,y,z,a) = R(x, y, z, -a) or -R(x,y,z,a) = R(-x,-y,-z,a)
	class SFRotation3d { double x, y, z, a; }; class SFRotation3f { float x, y, z, a; };
SFString	The SFString object contains single string encoded with the UTF-8 universal character set
	typedef SFString std::string;
SFTime	The SFTime specifies a single time value
	typedef SFTime double;
SFVec2{d|f}, SFVec3{d|f}, SFVec4{d|f}	The SFVec2d and SFVec3d objects specify a two-dimensional (2D) vector and a three-dimensional vector respectively. The SFVec4d field specifies a three-dimensional (3D) homogeneous vector. SFVec2d (or SFVec2f) is represented as a pair of double-precision (single-precision) floating-point values. So do SFVec3d (or SFVec3f) and SFVec4d (or SFVec4f)
	struct SFVec2d{ double v[2]; }; struct SFVec2f{ float v[2]; }; struct SFVec3d{ double v[3]; }; struct SFVec3f{ float v[3]; }; struct SFVec4d{ double v[4]; }; struct SFVec4f{ float v[4]; };

(To be continued)

(Table 3.1)

Types	Descriptions
SFTransform3{d|f}	The SFTransform3d and SFTransform3f objects specify the three-dimensional transformations by using double-precision and single-precision floating-point values. The field inside SFTransform3{d|f} is the same as that inside the node Transform of X3D standard. One SFTransform3{d|f} consists of multiple transformations, such as scaling, rotation and translation etc. Apply the transformation defined by SFTransform3{d|f} to an object means: ● Translate the object to the coordinates with the origin m_Center. The translation matrix defined by -m_Center is M_T(-m_Center). The center for scaling and rotation is m_Center. ● Rotate the object with the rotation matrix M_R(-m_ScaleOrientation) defined by -m_ScaleOrientation. ● Scale the object in the directions of three axes. The scaling transformation defined by m_Scale is M_S(m_Scale). ● Rotate the object to get back to the orientation before scaling. The rotation matrix M_R(m_ScaleOrientation) is defined by m_ScaleOrientation. ● Rotate the object with the rotation matrix M_R(m_Rotation) defined by m_Rotation. ● Translate the object to the origin of original coordinate with the translation matrix M_T(m_Center) defined by m_Center. ● Translate the object with the translation matrix M_T(m_Translation) defined by m_Translation. Given a 3D point v, applying the transformation defined by SFTransform3{d|f} is equivalent to making the computation as follows. v' = M_T(m_Translation) * M_T(m_Center) * M_R(m_Rotation) * M_R(m_ScaleOrientation) * M_S(m_Scale) * M_R(-m_ScaleOrientation) * M_T(-m_Center) * v

	struct SFTransform3f
	{
	SFVec3f m_Center;
	SFRotation3f m_Rotation;
	SFVec3f m_Scale;
	SFRotation3f m_ScaleOrientation;
	SFVec3f m_Translation;
	};
	struct SFTransform3d
	{
	SFVec3d m_Center;
	SFRotation3d m_Rotation;
	SFVec3d m_Scale;
	SFRotation3d m_ScaleOrientation;
	SFVec3d m_Translation;
	};

(To be continued)

(Table 3.1)

Types	Descriptions
SFQuaternion{d|f}	The SFQuaterniond and SFQuaternionf objects specify the quaternions by using double-precision and single-precision floating-point values. The quaternion is useful for representing rotation in a more mathematical way. The first value s is cos(theta/2), where theta is the rotation angle in radians. The remaining triple (x,y,z) is (vx*sin(theta/2), vy*sin(theta/2), vz*sin(theta/2)), where (vx, vy, vz) is the normalized vector representing the direction of rotation axis
	struct SFQuaterniond{ double s,x,y,z; }; struct SFQuaternionf{ float s,x,y,z; };
SFVariant	The SFVariant object specifies a variable that stores a value of any type. Developers can use the function T* any_cast<T>(SFVariant* v) to try to cast variable v to the class T. If the inside value in *v* can be cast to the class T, the function would return correct pointer, otherwise it would return NULL
SFNamedVariant	The SFNamedVariant object specifies a SFVariant-typed variable with a name

	struct SFNamedVariant
	{
	SFString m_Name;
	SFVariant m_Value;
	};

3.2.2 Multiple-Field Data Type

Corresponding to the single filed data types, the multiple-field data types are listed as follows: MFBool, MFColor, MFColorRGBA, MFDouble, MFFloat, MFInt32, MFNode, MFRotation, MFString, MFTime, MFVec{2|3|4}{d|f}, MFMatrix{3|4}{d|f}, MFTransform3{d|f}, MFQuaternion{d|f}, etc.

Prefix MF-means multiple field, which is the same as X3D's standard naming rules. Any data type prefixed with MF-is an array. The elements in the array have the same corresponding basic data type prefixed with SF-. For example,

```
namespace BasicType
{
    typedef MFBool   TMF<SFBool>;
    typedef MFColor  TMF<SFColor>;
    …
}
```

Here, BasicType::TMF<class T> is an array template, which supports any operation of std::vector.

3.2.3 Persistent Pointer: TAddress<>

In addition, there is a template of TAddress<class T>, which encapsulates a pointer to an object of class T and its ID. The data members are declared as follows:

```
template<class T>
class TAddress
{
     SFID ID;
     T * p;
public:
     T* operator ->(){...}
     T& operator *(){...}
// The other member functions are as follows.
...
}
```

TAddress overloads the operators of -> and *, and can be used as a pointer. If the value of TAddress::ID is –1, TAddress is invalid. TAddress is mainly used for inter-reference among entities. The most important advantage of TAddress is that the reference relationship among entities described by TAddress can be made persistent in files and host memory, since the ID is persistent. Therefore, we call the TAddress persistent pointer.

3.3 Basics of Scene Model

We first introduce the basic concepts of a scene model. Generally speaking, any scene model is composed of two parts of information. One part describes what entities are included in the scene. The other part answers how these entities are combined together.

A scene graph is the most common way to organize the data in the scene model. Basically all scene graphs are represented by the tree structure. Each scene has one root node as the ancestor of all nodes. Each node has zero, one or more children. But most nodes have only one parent node.

Generally speaking, in scene graphs each node is used to represent one entity in the scene. These entities are organized in a hierarchical tree structure. In most situations, middle nodes represent a set of objects and its child nodes represent one constituent part of this set. This hierarchy is called composition hierarchy. Such a hierarchical structure to describe the scene is in accordance with our understanding of the world.

We usually define the local coordinate for each node to describe the relative object locations more conveniently in this hierarchical structure. The object represented by the node is defined in this local coordinate. Besides, we need to

keep the transformation from this node's local coordinate to its parent node's local coordinate. Thus, a scene graph gives the transform hierarchy. Actually, a scene graph is also a kind of special composition hierarchy. A transform hierarchy is very useful for describing the movements of objects. In most situations, it is much easier to define the spatial location and movement of a constituent part in the local space than in the global space.

For example, if we use the solar system as the global space and the Sun as the origin, there is revolution around the Sun as well as rotation around its own axis for the Earth (Fig. 3.1). For a car moving on the Earth, its movement relative to the Earth is its movement on the sphere. If the car is abstracted to a point, we can regard it as rotating around the Earth's center. However, if we consider the car's movement in the global space, it would be very hard to describe this complex movement directly. It is very clear when using transform hierarchy to describe the movement.

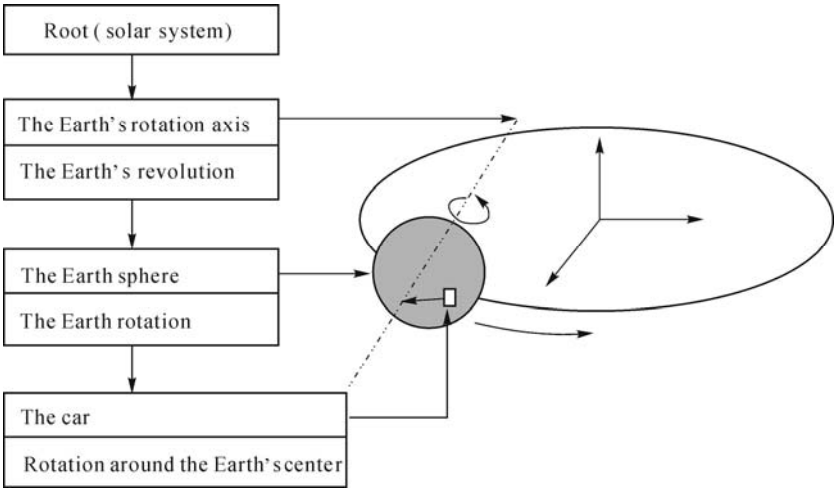

Fig. 3.1 An example of transform hierarchy

Besides the transform, we keep the bounding box of the objects in the node. In computer graphics, a bounding box hierarchy is a traditional data structure for rendering acceleration. During rendering, if the bounding box of one node is disjoint to the view frustum, it means that all objects represented by this node and its children nodes are not in the view frustum or at its border. This means we only need to perform one intersection computation for the node if its bounding box is not intersected with the view frustum. So we can perform view frustum culling to the objects in batches.

In Visionix, we represent basic entities in the scene and their relations in separate ways. We consider that the description of which entities are included in the scene is objective, while the description of their relations is subjective, since the relation reflects the scene modeler's viewpoint. Different scene modelers could have different relation descriptions for the same scene. Even for the same scene,

we may need different relation structures for different applications.

In the above example, the scene is composed of the Sun, the Earth and the car, while the relations among three objects are specified by applications. If we only need to show their belong-to relations, composition hierarchy is enough. If we need to describe their relative movements, we have to adopt transform hierarchy.

Besides the relations among entities, the relations between entities and the space are also very important. This is true not only because such a relation is very important to some applications, but also because it is very important for the rendering engine itself, since it is the basis for efficiency improvements in many computer graphics algorithms. In computer graphics, we usually build the relation between entities and the space by spatial index. The so-called spatial index subdivides a large space into many small sub-spaces hierarchically to form a space hierarchical tree. Each node of the tree corresponds to a sub-space and records the location relationship between each sub-space and the entities.

Based on the above understanding, the schemes adopted to design a scene manager in Visionix are as follows:

(1) All objects and their characteristics are kept in the structured container, scene storage. In Visionix, feature is the term used for objects' characteristics. One object is defined as the combination of many features. In Visionix there are rich feature types. Feature can be nested, i.e., one feature can be combined by many child features. All instances of one feature are stored in an object table. The object tables are stored in the scene storage.

(2) We adopt a hierarchical structure to describe the relations of features. The scene manager can contain multiple hierarchical structures, called Visionix Scene Graph, or Scene Graph for short.

(3) We adopt a hierarchical structure to describe the relations between feature and space. The scene manager can contain multiple hierarchical structures, called Visionix Space Index, or Space Index.

In Visionix scene manager, one scene model is composed of one scene storage, multiple Scene Graphs and multiple Spatial Indexes.

3.4 Entity

In Visionix, IEntity is the base class of the objects that should be persistent and allows inter-referencing or being inter-referenced. IEntity is declared as follows:

```
class IEntity
{
public:
     virtual SFInt32 EntityType() const = 0;
     virtual void Accept(IVisitor* pVisitor) = 0;
// The other member functions are omitted.
        …
```

```
protected:
     SFID m_ID;
};
```

The field m_ID is the unique identifier of an IEntity instance. If one TAddress pointer points to this class's instance, the ID in the TAddress pointer is the same as m_ID in the IEntity instance.

Each concrete subclass of IEntity has to implement the virtual function,

```
SFInt32 EntityType () const,
```

which returns the concrete class type of the instance.

We adopt the visitor design pattern to realize visits to scene models in Visionix. The function,

```
SFInt32 Accept (IVisitor* pVisitor),
```

provides visitors with a unified entry point.

3.5 Feature

We refer to the standards in X3D to define features in Visionix with some tailoring and revisions, according to our experiences in developing practical applications.

All types of features form the class hierarchy, feature hierarchy, which is similar to the node hierarchy in X3D. The base class is class IFeature. All other features are derived from IFeature. Here we only introduce the most important feature class for clarity, whose class hierarchy is shown in Fig. 3.2.

All classes whose class names are prefixed with "I" are abstract classes in Visionix. The character "I" means Interface.

3.5.1 IAttributedObject and IFeature

The class IAttributedObject is the base class representing the objects which have attributes. Its data members are declared as follows:

```
class IAttributedObject
{
protected:
    TMF< TAddress<IAttribute> > m_AttriSet;
…
}
```

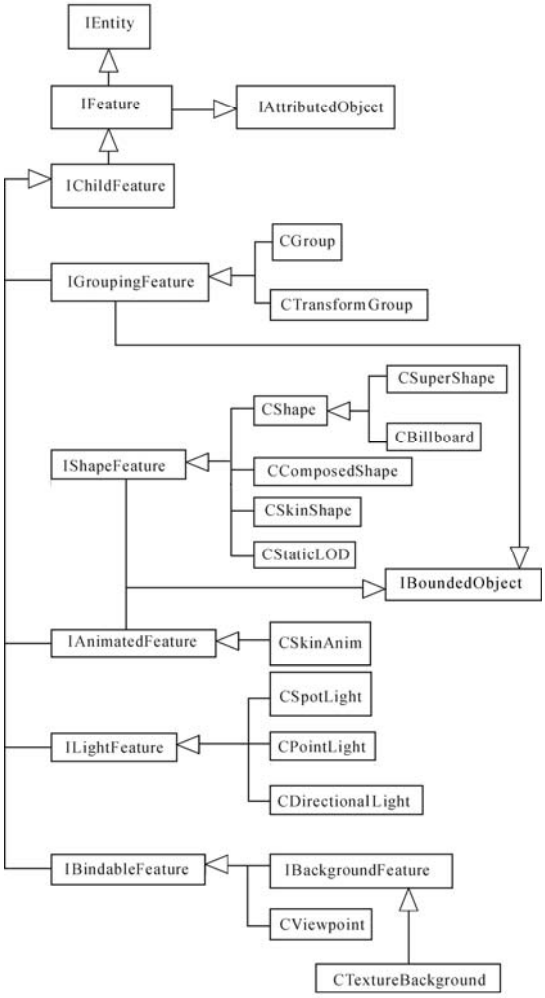

Fig. 3.2 The class hierarchy of features

IAttributedObject contains an array of TAddress pointers which point to IAttribute instances. IAttribute is the base class of various attributes and, as an abstract interface, it has no member variable. Some Attributes are pre-defined in Visionix. Developers can define a concrete subclass of IAttribute to meet their demands.

IFeature inherits both IEntity and IAttributedObject to act as the base class of all Features. IFeature itself does not introduce new data members.

```
class IFeature: public IEntity, public IAttributedObject
{
…
}
```

3.5.2 IBoundedObject

The objects occupying bounded space are all derived from IBoundedObject. The data members in IBoundedObject are declared as follows:

```
class IBoundedObject
{
protected:
    SFVec3f m_BBoxCenter;
    SFVec3f m_BBoxSize;
};
```

IBoundedObject actually defines an axis aligned bounding box (AABB). It contains two data members, m_BBoxCenter is the center of the AABB, and m_BBoxSize is the size of the AABB.

IShapeFeature, IGroupingFeature and IAnimatedFeature are derived from IBoundedObject. In addition, the spatial index node is also derived from IBoundedObject, which will be introduced later.

3.5.3 IChildFeature

Only the feature inheriting IChildFeature can be a child feature, which is the constituent part of another feature.

Any feature derived from IGroupingFeature represents a feature group, consisting of several child features. Since it is also derived from IChildFeature, one feature group, or say one IGroupingFeature instance, can be the constituent part of another bigger feature group.

IChildFeature and IGroupingFeature form the composition design pattern as shown in Fig. 3.3.

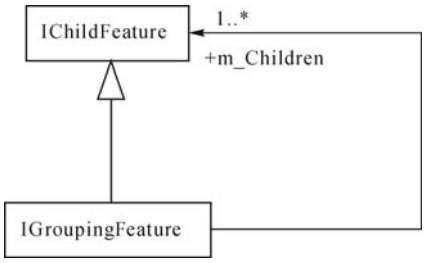

Fig. 3.3 The composition of IGroupingFeature and IChildFeature

Besides IGroupingFeature, there are four child features in Visionix:

(1) Any class inheriting IShapeFeature represents the feature that has some spatial location and is of static geometry and appearance. Most features consisting

of a scene are the concrete classes of IShapeFeature.

(2) Any class inheriting IAnimatedFeature represents the feature that has some spatial location, but is of dynamic geometry or appearance.

(3) Any class inheriting ILightFeature represents light sources with spatial locations.

(4) Any class inheriting IBindableFeature represents such features that there is, at most, one feature of this class valid at any run-time. Here we adopt the same way to define the class as that for X3DBindableNode in X3D.

3.5.4 Subclasses of IGroupingFeature

IGroupingFeature derives from both IFeature and IBoundedObject. IGroupingFeature is declared as follows:

```
class IGroupingFeature:public IChildFeature, public IBoundedObject
{
protected:
     TMF<TAddress<IChildFeature>>m_Children;
…
};
```

IGroupingFeature has two concrete classes: CGroup, CTransformGroup. The class diagram is shown in Fig. 3.4.

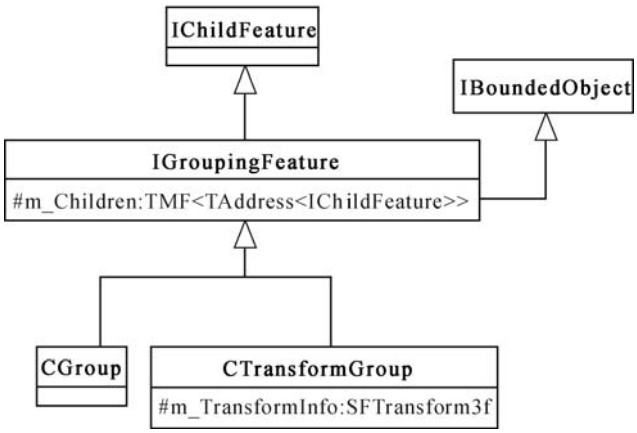

Fig. 3.4 The class hierarchy of IGroupingFeature's family

3.5.4.1 CGroup

CGroup is the basic IGroupingFeature. It does not have any new additional data members. It is just a direct implementation of abstract class IGroupingFeature.

3.5.4.2 CTransformGroup

CTransformGroup is a bit more complex than CGroup. Besides the composition relation, it also introduces coordinate transformation m_TransformInfo (please refer to the introduction of SFTrasform3f in Section 3.2). The use of m_TransformInfo is to transform the local coordinate of CTransformGroup node to its parent node's coordinate. We can easily build the inner transform hierarchy for features with CTransformGroup.

In order to be compatible with the Transform node in X3D, we define a quintuple by SFTransform3f instead of directly defining a 4×4 transformation matrix in CTransformGroup node.

3.5.5 Subclasses of IShapeFeature

Most of all the entities in a scene are the features inheriting IShapeFeature. To meet the requirements of different applications, there are several kinds of Shape features defined: CShape, CBillboard, CSkinShape, CSuperShape and CComposedShape. These class hierarchal relations are shown in Fig. 3.5.

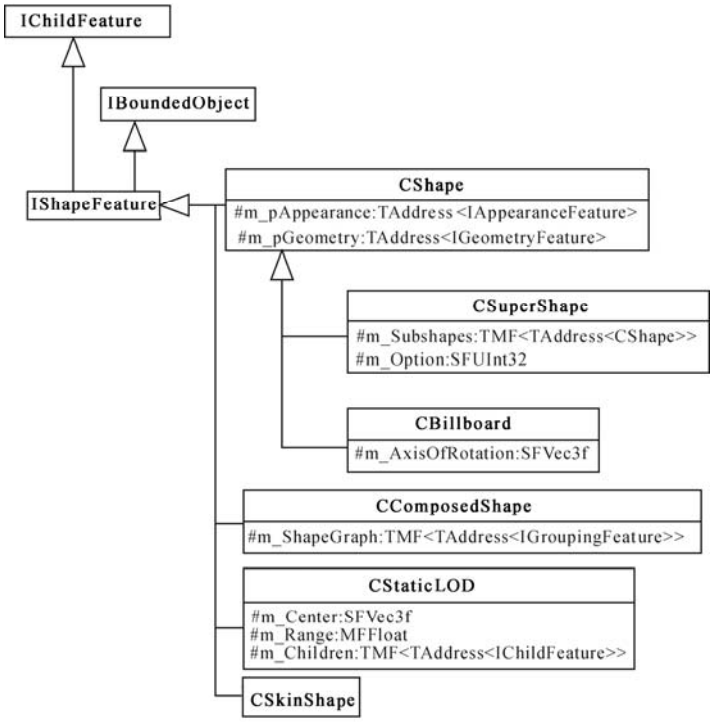

Fig. 3.5 The class hierarchy of IShapeFeature's family

3.5.5.1 CShape

CShape is the basic shape feature which contains two data members, m_pGeometry and m_pAppearance. m_pGeometry points to an IGeometryFeature instance, which represents the geometry of the shape feature. m_pAppearance points to an IAppearanceFeature instance, which describes the appearance, such as texture and material. We will introduce IGeometryFeature and IAppearanceFeature in later sections.

3.5.5.2 CBillboard

CBillboard is a special kind of Shape with the additional direction vector m_AxisOfRotation. This vector actually defines a rotation axis and axial cross section.

Rotation axis: The vector passes through the object local coordinate frame's origin and takes m_AxisOfRotation as the positive direction.

Axial cross section: The plane takes the axial positive direction as positive normal and passes through the local coordinate frame's origin.

With rotation axis and axial cross section, what CBillboard represents is expressed as follows. During rendering we need to rotate the object's local coordinate properly around the rotation axis, in order to make the projection of the vector (from local coordinate frame's origin to viewpoint) on the axial cross section coincide with the projection of the *Z* axis of the local coordinate.

In most situations, the shape of CBillboard is a planar polygon. The *Z* axis of the local coordinate is set to be the normal of this planar polygon. This leads to the effect that the polygon always rotates around the axis when the viewpoint changes. Thus the projection of the plane's normal on the cross section always points to the projection of the viewpoint on the cross section. It is worth noting that CBillboard can only guarantee that the geometry's projection always points to the viewpoint's projection. It can not guarantee that the geometry always points to the viewpoint in 3D space (unless the viewpoint is located at the cross section).

Since the CBillboard feature possesses the above characteristic of rotation, it is usually used to approximately represent an axially symmetric or semi-axially symmetric object. The most widely-used example is the representation of trees by CBillboard features. The tree is modeled as a planar textured quadrilateral, where texture contains alpha values. The texture shows the front view of the tree, and the alpha values are used to differentiate the tree from the background. During rendering, alpha culling is used to render the tree's complex contour. The drawback of this method is that when the viewpoint is far away from the axial cross section and near to the rotation axis, the observer would find the tree is just a planar polygon.

Note that the CBillboard feature defined in Visionix differs a bit from Billlboard defined in X3D, though their basic concepts are accordant. In Visionix, CBillboard is a kind of shape feature while Billboard in X3D is a kind of group. It is more convenient to understand and process CBillboard if it is a single shape feature,

since one instance of a CBillboard can only represent one object. In X3D, Billboard is treated as group nodes for the purpose of applying the rotation characteristic of Billboard to all its children nodes. The geometry and appearance of the object corresponding to Billboard is represented by its children nodes. This scheme is more suitable for representing objects with a rotation property of much more complex geometry and appearance. According to our experience, the objects with the rotation property described by Billboard are mainly simple polygons. The appearance is just a texture. So there is no need to adopt the definition of Billboard in X3D.

3.5.5.3 CSuperShape

In many situations there are objects with huge volumes in 3D scenes. These objects impact the efficiency of view culling—even if only a small part of the object is visible, we have to render the entire object. This would severely reduce the rendering speed. The normal way to solve this problem is to divide these objects into a set of sub-objects. Such division is performed according to the space occupied by the object, so the sub-objects themselves do not have any logical or semantic meanings. If we want to give attention to both rendering efficiency and the object's semantic meaning, a data structure is needed to keep the original object and the divided sub-objects simultaneously.

So in Visionix we introduce CSuperShape to solve this problem. The declaration of the data member of the class CSuperShape is listed as follows:

```
class CSuperShape:public CShape
{
public:
  enum Enum_OptMask{
     PARTITIONED = 1,
     SUBDIVIDED =2
  };
protected:
  TMF<TAddress<CShape>> m_Subshapes;
  SFUInt32    m_Option;
};
```

On the basis of CShape, we add a container to store divided sub-shapes inside CSuperShape. By this means, we achieve the goal of storing both the original object and subdivided objects.

It is worth noting that since we adopt TAddress pointers to make inter-reference of features or entities, there is only one copy of the same data in divided subshapes and the original shape. Therefore, there is no storage space wasted.

CSuperShape can not only be used to represent the subshapes of huge-volume objects, but also the objects that need subdivision for other reasons. One common situation is the object with multiple materials. Take a triangular mesh as an example. If some triangles on this mesh's surface have red diffuse material, while others have blue specular material, we cannot use one CShape instance to

represent such a mesh. The traditional method is to adopt multiple CShape instances. This would lead to the loss of the semantic meaning of the original model. CSuperShape can solve this problem satisfyingly by dividing the object according to its material so that each subshape has only one material.

If we start from the definition of the class, any subshape of a CSuperShape instance can also be a CSuperShape instance. Only if there are two different divisions needed to be represented simultaneously, we would need the above complex representation. For example, if the object with multiple materials also needs spatial division, a two-level hierarchical structure of CSuperShape is required.

m_Option describes some Boolean values in CSuperShape. It is used together with the enumeration types defined by Enum_OptMask.

m_Option& PARTITIONED: If it is 1, this means subshapes come from the cut of the geometry in CSuperShape. New vertices are generated in this situation. If it is 0, this means subshapes come from the classification of the geometry in CSuperShape. No new vertices would be produced.

m_Option&SUBDIVIDED: If it is 1, this means the geometry of the subshape comes from the subdivision of the geometry in SuperShape. New vertices and topology are produced. Otherwise, there is no subdivision.

We give some examples to explain the meanings of these two values:

Example 1: The object M is composed of two triangles A and B. If these two triangles are subdivided to get two triangular meshes M(A) and M(B), M(A) and M(B) form two subshapes. In this situation, m_Option& PARTITIONED is set to be 0 and m_Option&SUBDIVIDED is set to be 1.

Example 2: The object M is composed of two triangles A and B. If we cut triangle A and triangulate two sub-regions of A, we get three triangles A1, A2 and A3. Triangle B is not cut. Triangles A1 and A2 form a subshape. Triangles A3 and B form the other subshape. In this situation, m_Option& PARTITIONED is set to be 1 and m_Option&SUBDIVIDED is set to be 0.

Example 3: The object M is composed of two triangles A and B. If we cut triangle A and triangulate two sub-regions of A, we get three triangles A1, A2 and A3. Triangle B is not cut. Then triangles A1, A2, A3 and B are subdivided. In this situation, m_Option& PARTITIONED is set to be 1 and m_Option&SUBDIVIDED is set to be 1. If we perform the subdivision before the cut, the values of these two properties are also set to be 1.

3.5.5.4 CComposedShape

The declaration of the data member of the class CComposedShape is listed as follows:

```
class CComposedShape:public IShapeFeature
{
protected:
     TMF<TAddress<IGroupingFeature>> m_ShapeGraph;
};
```

The data member m_ShapeGraph in CComposedShape contains an array of TAddress pointers pointing to IGroupingFeature instances. Its name shows that m_ShapeGraph is a hierarchical structure—a tree structure composed of subclasses of IGroupingFeature and IChildFeature (if we consider the inter-references among these subclasses, the tree is actually a non-cyclic digraph). The type of the middle node of the tree is the subclass of IGroupingFeature. The type of the leaf node is the subclass of IChildFeature.

The concrete composition of CComposedShape is flexible. However, such flexibility would bring complexity to the applications. In order to reduce the complexity of data types, we need to limit CComposedShape properly, according to the application. Such limitation is called a Scene Model Scheme which will be introduced later. In most situations, the type of the leaf node is CShape. However, if we start from class definition, we do not exclude the possibility to adopt other subclasses of IChildFeature as the type of leaf node in some applications.

Note that CComposedShape is derived from IShapeFeature instead of CShape, while CSuperShape is derived from CShape. Hence, CSuperShape actually gives two representative models for one object—one is the non-divided model and the other is the model after object division. CComposedShape only gives one representative model. It describes how this model is composed of multiple sub-models in detail.

3.5.5.5 CStaticLOD

```
class CStaticLOD:public class IShapeFeature
{
protected:
      TMF<TAddress<IChildFeature>> m_Children;
      MFFloat                      m_Range;
      SFVec3f                      m_Center;
}
```

The data members in CStaticLOD are the same as those of LOD node in X3D. However, we let it derive from IShapeFeature instead of IGroupingFeature, which is different from X3D, due to the fact that the child features contained in CStaticLOD do not have a semantic composition relationship.That is to say, all child features in CStaticLOD depict the same object in the scene with a different level of detail.

The CStaticLOD class specifies various levels of detail for a given object, and provides hints that allow programs to choose the appropriate detail level of the object according to the distance from the viewpoint The m_Children field contains a list of TAddress<IChildFeature>, which represents the same object at different levels of detail, ordered from the highest level to the lowest level. The m_Range field specifies the distances at which one can switch between the levels. The m_Center field specifies the centre of the CStaticLOD feature for distance calculations.

3.5.5.6 CSkinShape

CSkinShape class represents the dynamic objects which have skeletal animation. We will introduce this class when we introduce IAnimatedFeature subclass.

3.5.6 IAnimatedFeature

So far, Visionix only provides skeletal animation based dynamic features—CSkinAnim. The related class diagram is shown in Fig. 3.6. Note that the associate relationship in the class diagram is realized by TAddress<target class>. A multi-associate relationship is realized by TMF<TAddress<target class>>.

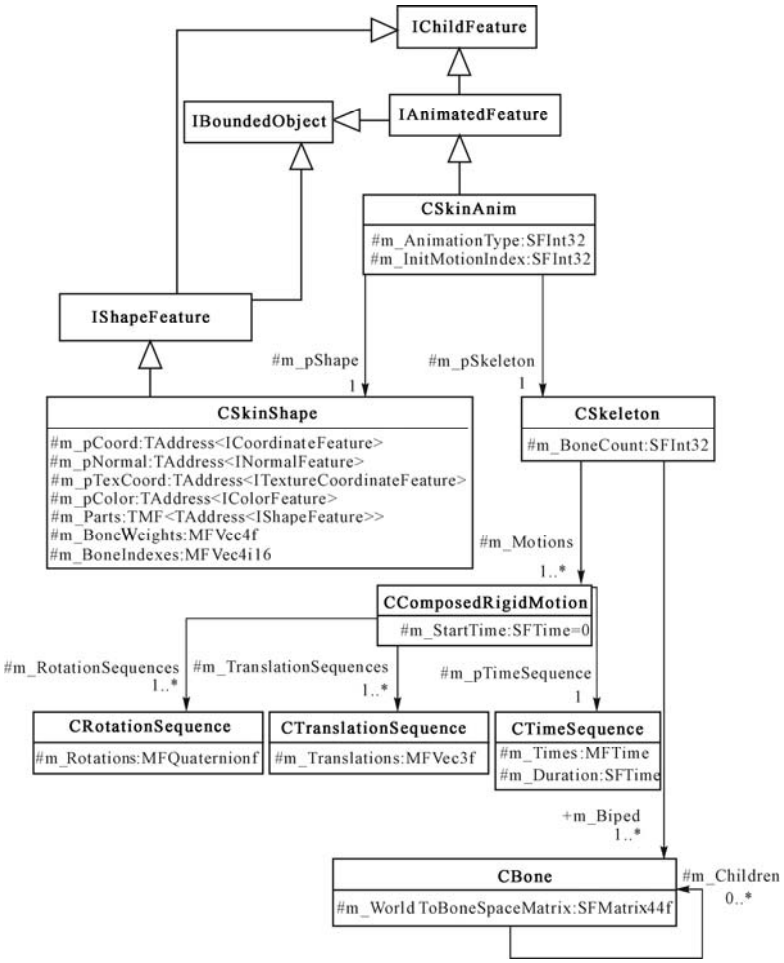

Fig. 3.6 The classes related to IAnimatedFeature

The data members of m_pShape and m_pSleleton point to the instances of CSkinShape and CSkeleton respectively. CSkinShape is a special kind of Shape, and describes the skin shape of the skeleton. CSkeleton describes the skeleton and its movements.

The data member m_Biped in CSkeleton points to the root node of the hierarchical structure consisting of CBone instances. This structure describes the skeleton composition. Besides CBone::m_Children containing an array of TAddress<CBone> pointing to the children bone node, there is a matrix m_WorldToBoneSpaceMatrix, which transforms the world coordinate frame to the bone's local coordinate frame. Such a transformation is to make more convenient the implementation of skinning animation. To compute skinning animation, the first step is to transform the coordinates of vertices of the skinning mesh to the world coordinate frame. Then, according to the binding relation between the vertices and the bones, these vertices are transformed to their corresponding bones' local coordinate frames by multiplying m_WorldToBoneSpaceMatrix of the bones'. After that, the correct animation effect can be realized by further multiplying the bones' animation matrixes to the vertices.

The data member m_Motions in CSkeleton is the array of TAddress <CComposedRigidMotion>. Each CComposedRigidMotion instance describes one set of skeleton movement, which means that CSkeleton::m_Motions describes multiple sets of skeleton movements.

CComposedRigidMotion is the subclass of IMotionFeature. IMotionFeature is the abstract class to describe objects' movement. The data member CComposedRigidMotion::m_pTimeSequence points to the time sequence CTimeSequence, which records the duration time of the whole time sequence (denoted by m_Duration) and the timing of some key frames during the duration time (denoted by m_Times). The data members m_RotationSequences and m_TranslationSequences in CComposedRigidMotion record the rotation sequences and translation sequences respectively. The time sequence in CComposedRigidMotion is valid for all rotation and translation sequences. This means that for the *i*-th key frame at time t_i, the *i*-th element in any rotation sequence is the object's rotation quaternion at time t_i, and the *i*-th element in any translation sequence is the object's translation vector at time t_i. The data member CComposedRigidMotion::m_StartTime records the starting time of the movement with the default value of 0.

For each CComposedRigidMotion instance in CSkeleton::m_Motions, the rotation sequence is paired with the translation sequence. Each rotation-translation sequence pair describes one movement of one bone. Therefore, the number of the bones in a CSkeleton instance should be equal to the number of rotation-translation sequence pairs in each CComposedRigidMotion instance in the Skeleton instance.

The data member CSkinAnim::m_pShape points to a CSkinShape instance. As we have stated before, CSkinShape is the subclass of IShapeFeature.

```
CSkinShape:public IShapeFeature
{
protected:
     TAddress<ICoordinateFeature>           m_pCoord;
```

```
    TAddress<INormalFeature>                m_pNormal;
    TAddress<ITextureCoordinateFeature>     m_pTexCoord;
    TAddress<IColorFeature>                 m_pColor;
    TMF<TAddress<IShapeFeature>>            m_Parts;
    MFVec4f                                 m_BoneWeights;
    TMF<SFInt16[4]>                         m_BoneIndexes;
}
```

Similar to CSuperShape, CSkinShape has the data member m_Parts which contains an array of TAddress<IShapeFeature> pointing to subshapes. It means that the shape of the skinning mesh is represented by these subshapes. CSkinShape explicitly separates the reused coordinates, normal vectors, texture coordinates and colors from the parts. They are referenced in the parts. In applications, these parts are usually yielded according to the materials on the mesh surface.

The data member m_BoneIndexs contains indices of bones influencing each vertex. Since bones are organized like a tree in the skeleton, bone indices are given in depth-first order. In Visionix, we set a limit so that each vertex will be influenced by no more than four bones' positions. So the four bone indices influencing the *i*-th vertex m_pCoord->m_Point[*i*] are stored in the four dimensional vectors in m_BoneIndexes[*i*]. The bones are numbered from 1. The value 0 represents invalid bones. If the number of influencing bones of one vertex is less than four, we use 0 to fill in unused bits. The data member m_BoneWeights records the influencing weights of the bones to the vertices. The weights correspond to the bones, which is of the same correspondence between indices and vertices in m_BoneIndex. For *i*-th vertex m_pCoord->m_Point[*i*], its location would be influenced by at most four bones with indices m_BoneIndexes[*i*][0], m_BoneIndexes[*i*][1], m_BoneIndexes[*i*][2] and m_BoneIndexes[*i*][3]. The related four weights of bones are m_BoneWeights[*i*][0], m_BoneWeights[*i*][1], m_BoneWeights[*i*][2] and m_BoneWeights[*i*][3].

3.5.7 Subclasses of ILightFeature

There are three kinds of lights defined in Visionix, which are spot light, point light and directional light. The corresponding classes are CSpotLight, CPointLight and CDirectionalLight respectively (Fig. 3.7). The definitions of these classes are the same as SpotLight, Point and Directional Light in X3D.

3.5.8 Subclasses of IBindableFeature

For subclasses of IBindableFeature, IBackgroundFeature is used to describe the surroundings of the scene. CTextureBackground represents the surrounding background using a textured box (Fig. 3.8). We can pre-store multiple backgrounds for the scene, but at any time only one background is selected.

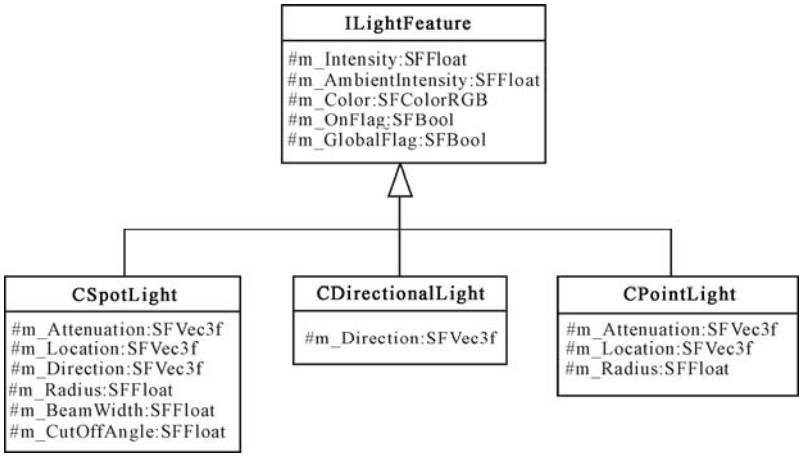

Fig. 3.7 The subclasses of ILightFeature

CTextureBackground
#m_SkyAngle: MFFloat
#m_GroundAngle: MFFloat
#m_Transparency: MFFloat
#m_Size: SFFloat
#m_SkyColor: MFColorRGB
#m_GroundColor: MFColorRGB
#m_pLeftTexture: TAddress<ITextureFeature>
#m_pFrontTexture: TAddress<ITextureFeature>
#m_pBottomTexture: TAddress<ITextureFeature>
#m_pBackTexture: TAddress<ITextureFeature>
#m_pTopTexture: TAddress<ITextureFeature>
#m_pRightTexture: TAddress<ITextureFeature>

Fig. 3.8 Class CTextureBackground

Just as the name indicates, CViewpoint is used to describe the viewpoint information. It is obvious that there is only one viewpoint valid at any time for one rendering engine. The definition of CViewpoint is shown Fig. 3.9. The definitions of data members can be referred to the Viewpoint node in X3D.

CViewpoint
#m_bJump: SFBool
#m_FieldOfView: SFFloat
#m_Position: SFVec3f
#m_CenterOfRotation: SFVec3f
#m_Orientation: SFRotation3f
#m_Description: SFString

Fig. 3.9 Class CViewpoint

3.5.9 IGeometryFeature

IGeometryFeature is the base class for all classes describing geometries. Its concrete subclasses are used to describe the geometry in CShape class. The class hierarchy of IGeometryFeature is shown in Fig. 3.10. So far there are three types of Geometry Features defined in Visionix: IComposedGeometryFeature, ICurveFeature and ITextFeature.

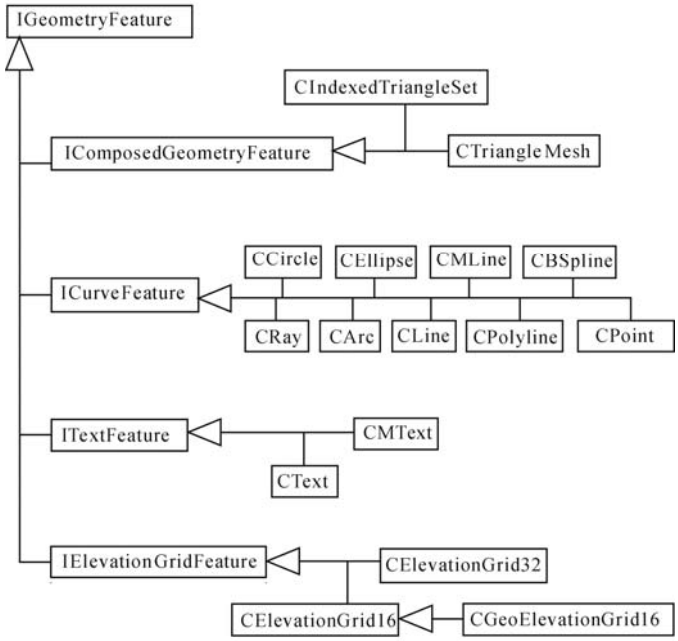

Fig. 3.10 The class hierarchy of IGeometryFeature's family

3.5.9.1 IComposedGeometryFeature

The most common 3D solid geometry in Visionix is represented by CIndexedTriangleSet, which is derived from IComposedGeometryFeature. From its name we can see that IComposedGeometryFeature is composed of some basic geometry elements—geometric property. There are many basic geometry elements defined in X3D standard, such as Box, Cone, Sphere and Cylinder. Since these geometry elements can be transformed to CIndexedTriangleSet, we do not define such geometry elements any more in Visionix.

CTriangleMesh is another very important geometry representation, and is designed for efficient mesh editing. Compared with CIndexedTriangleSet, CTriangleMesh consists of multiple indices, which represents the complete topology information of triangle mesh.

Fig. 3.11 gives the data members of IComposedGeometryFeature and its subclasses.

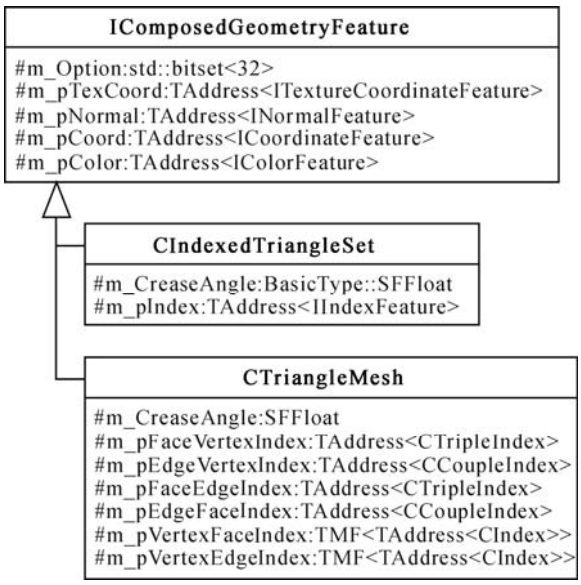

Fig. 3.11 Class IComposedGeometryFeature and the subclasses

3.5.9.2 CIndexedTriangleSet and Geometric Properties

CIndexedTriangleSet contains several TAddress pointers, pointing to some geometric property features, such as normal, coordinate, color and texture coordinate. Fig. 3.12 gives the base class and its subclasses of geometric property feature.

(1) ICoordinateFeature represents vertices' coordinates. CCoordinate as its concrete class represents 3D coordinates.

(2) ITextureCoordinateFeature represents vertices' texture coordinates. The three concrete subclasses of ITextureCoordinateFeature are CTextureCoordinate, CTextureCoordinate3D and CMultiTextureCoordinate. They are respectively used to represent single 2D texture coordinates, single 3D texture coordinates and multi-texture coordinates. CTextureCoordinate contains m_Point the array of 2D vectors. CTextureCoordinate3D contains m_Point the array of 3D vectors. CMultiTextureCoordinate contains the array of TAddress <ITextureCoordinateFeature> pointing to ITextureCoordinateFeature instances, each of which represents the texture coordinates corresponding to one level texture in multi-textures. The nested use of CMultiTextureCoordinate is not allowed in Visionix, which means that any CMultiTextureCoordinate instance is forbidden to appear in m_TexCoord array.

(3) INormalFeature represents vertices' normal. CNormal stores the vertex normal in 3D vector array.

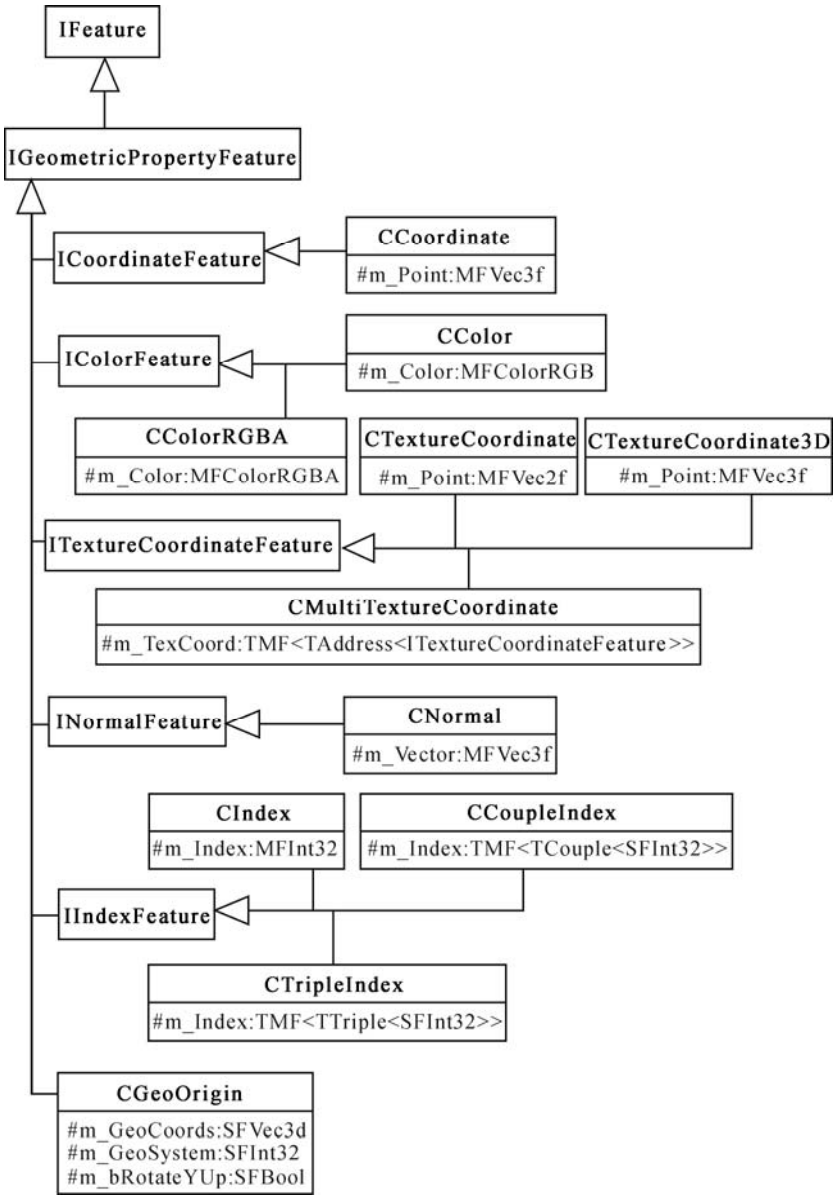

Fig. 3.12 The class hierarchy of IGeometricPropertyFeature's family

(4) IColorFeature represents vertices' colors. CColor and CColorRGBA are two subclasses of IColorFeature. Each element of the array m_Color in CColor represents (Red, Green, Blue). Each element of the array m_Color in CcolorRGBA represents (Red, Green, Blue, Alpha). Alpha represents opacity.

(5) IIndexFeature represents the topology of the geometry and the corresponding

relation between vertices and colors, texture coordinates and normal. Strictly speaking, it is more proper to be called topological property instead of geometric property for IIndexFeature. Here we treat it as a geometric property in order to reduce new base classes. Besides, it is more understandable for developers to use it as a geometric property. CIndex actually is an array of integers. CTripleIndex is an array of triples of integers, while CCoupleIndex is an array of couples of integers. The templates of TTriple<T> and TCouple<T> are listed as follows.

```
template <class T>
struct TTriple {
     T data[3];
     T& operator [](int i);
     const T& operator [](int i) const;
};

template <class T>
struct TCouple{
     T data[2];
     T& operator [](int i);
     const T& operator [](int i) const;
};
```

(6) CGeoOrigin represents the local coordinate system of a terrain grid. For more details, please refer to the IElevationGridFeature section.

With these concepts, we can better understand IComposedGeometryFeature and CIndexedTriangleSet classes. IComposedGeometryFeature class composes four geometric properties including coordinates, normal, colors and texture coordinates. CIndexedTriangleSet represents the vertices connection by CIndex. In CIndex, every three integers correspond to three vertices' indices of one triangle. According to the indices, the corresponding vertex coordinates, normal, colors and texture coordinates can be retrieved in the geometric properties. The following pseudo-code shows how to retrieve the vertex coordinates, normal, colors and one-level texture coordinate for three vertices of the first triangle.

```
// three vertices′ coordinates
m_pCoord->m_Point[m_pIndex->m_Index[0]],
m_pCoord->m_Point[m_pIndex->m_Index[1]],
m_pCoord->m_Point[m_pIndex->m_Index[2]].

// three normal vectors
m_pNormal->m_Point[m_pIndex->m_Index[0]],
m_pNormal->m_Point[m_pIndex->m_Index[1]],
m_pNormal->m_Point[m_pIndex->m_Index[2]].
```

```
// three color vectors
m_pColor->m_Color[m_pIndex->m_Index[0]],
m_pColor->m_Color[m_pIndex->m_Index[1]],
m_pColor->m_Color[m_pIndex->m_Index[2]].

// three vertices' texture coordinates
m_pTexCoord->m_Point[m_pIndex->m_Index[0]],
m_pTexCoord->m_Point[m_pIndex->m_Index[1]],
m_pTexCoord->m_Point[m_pIndex->m_Index[2]].
```

In Visionix, geometric properties including coordinate, color, texture coordinate and normal are defined separately in order to reuse the geometric properties as much as possible to reduce the wasted space.

IComposedGeometryFeature has a bit set m_Option, each bit of which is defined as follows:

Lowest order bit (the zeroth bit), CCW: If it is 1, this means that the triangle vertices are arranged in counter-clock wise order. Otherwise, they are in clock wise order.

The first bit, COLORPERVERTEX: If it is 1, this means that colors are stored for each vertex. Otherwise colors are stored for each triangle. This flag bit is invalid for CIndexedTriangleSet, since all geometric properties in CIndexedTriangleSet are stored for each vertex. This flag bit is for compatibility of other classes derived from IComposedGeometryFeature.

The second bit, NORMALPERVERTEX: If it is 1, this means the normal is stored for each vertex. Otherwise, the normal is stored for each triangle. This flag bit is invalid for CIndexedTriangleSet since all geometric properties in CIndexedTriangleSet are stored for each vertex. This flag bit is for compatibility of other classes derived from IComposedGeometryFeature.

The third bit, SOLID: If it is 1, this means the corresponding geometry has a closed surface. Otherwise, it is non-closed. During rendering, if this bit is 1 back face culling is automatically used.

The fourth bit, CONVEX: If it is 1, this means that the object is convex. Otherwise, it is concave. This is useful for collision detection.

The data member m_CreaseAngle in CIndexedTriangleSet indicates a crease critical angle in radians. This critical angle is used to compute the normal and further to control the surface's smooth transition under lighting.

For two triangles A and B sharing one edge (which means they share two vertices $\boldsymbol{v}_1$, $\boldsymbol{v}_2$), if the angle between these two triangles is less than or equal to m_CreaseAngle, the normal of two shared vertices is set to be the average of the normal of these two triangles, $\boldsymbol{n}=0.5(\boldsymbol{n}_A+\boldsymbol{n}_B)$. (Actually the normal $\boldsymbol{n}$ needs to be normalized after the above computation. We ignore this for the convenience of expression, which is the same for the following computation.) This means for triangle A, the normal of its vertices $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ is set to be $\boldsymbol{n}$. For triangle B, the normal of its vertices $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ is also set to be $\boldsymbol{n}$. If the angle between these two triangles is larger than m_CreaseAngle, the normal of two shared vertices is set to

be the normal of their corresponding triangles. This means that for triangle A, the normal of its vertices $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ is set to be $\boldsymbol{n}_A$. For triangle B, the normal of its vertices $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ is set to be $\boldsymbol{n}_B$.

Assume that three triangles A, B and C share one vertex $\boldsymbol{v}$. If the angle between A and B, the angle between A and C, and the angle between B and C are all less than m_CreaseAngle, the normal $\boldsymbol{n}$ of $\boldsymbol{v}$ is set to be the average of three normal vectors of A, B and C, $\boldsymbol{n} = (\boldsymbol{n}_A, \boldsymbol{n}_B, \boldsymbol{n}_C)/3$. If they are all larger than m_CreaseAngle, the normal of vertex $\boldsymbol{v}$ in triangle A is set to be $\boldsymbol{n}_A$. The normal of vertex $\boldsymbol{v}$ in triangle B is set to be $\boldsymbol{n}_B$. The normal of vertex $\boldsymbol{v}$ in triangle C is set to be $\boldsymbol{n}_C$. If the angle between A and B, the angle between B and C are less than m_CreaseAngle, the angle between A and C is larger than m_CreaseAngle, the normal of $\boldsymbol{v}$ in B is set to be $(\boldsymbol{n}_A, \boldsymbol{n}_B, \boldsymbol{n}_C)/3$, the normal of $\boldsymbol{v}$ in A is set to be $(\boldsymbol{n}_A + \boldsymbol{n}_B)/2$, and the normal of $\boldsymbol{v}$ in C is set to be $(\boldsymbol{n}_B + \boldsymbol{n}_C)/2$.

For vertices shared by more than three triangles, the triangles can be classified to some clusters according to their neighboring and angle information. If the number of clusters is greater than or equal to three, the normals of vertices in the cluster are set to be the average of all triangles' normals. If there are only two clusters, they can be processed in a similar way as the situation of three triangles sharing one vertices described in the above paragraph. Of course, the better way is to compute the weighted mean according to angles between triangles.

Since the shared vertex in one cluster has the same normal for multiple triangles, the lighting of this shared vertex is the same. So the lighting transition is smooth around the boundaries of multiple triangles. However, triangles of different clusters have an obvious lighting change across the boundaries.

3.5.9.3 CTriangleMesh

```
class CTriangleMesh:public IComposedGeometryFeature
{
protected:
    SFFloat                   m_CreaseAngle;
    TAddress<CTripleIndex>    m_pFaceVertexIndex;
    TAddress<CCoupleIndex>    m_pEdgeVertexIndex;
    TAddress<CTripleIndex>    m_pFaceEdgeIndex;
    TAddress<CCoupleIndex>    m_pEdgeFaceIndex;
    TMF<TAddress<CIndex>>     m_pVertexFaceIndex;
    TMF<TAddress<CIndex>>     m_pVertexEdgeIndex;
};
```

The class declaration of CTriangleMesh is listed as above. Several index arrays keep the relationship among triangle faces, edges and vertices of the mesh. Each triple in the index array pointed by m_pFaceVertexIndex stands for a triangle face, and each element in the triple is an index of vertex, which is stored in m_pCoord->m_Point. The three vertices of the *i*-th triangle, ($\boldsymbol{v}0$, $\boldsymbol{v}1$, $\boldsymbol{v}2$), can be

accessed by the following codes.

```
TTriple<SFInt32> face = m_pFaceVertexIndex->m_Index[i];
v0 = m_pCoord->m_Point[face[0]];
v1 = m_pCoord->m_Point[face[1]];
v2 = m_pCoord->m_Point[face[2]];
```

Every couple in the index array pointed by m_pEdgeVertexIndex represents an edge of the triangle mesh. The two vertices of the *j*-th edge, (***v***0, ***v***1) can be visited by the following codes.

```
TCouple<SFInt32> edge = m_pEdgeVertexIndex->m_Index[j];
v0 = m_pCoord->m_Point[edge[0]];
v1 = m_pCoord->m_Point[edge[1]];
```

By using m_pFaceEdgeIndex, we solve the query about which edges a triangle consists of. For the *i*-th triangle, the indices of its three edges are kept in the triple m_pFaceEdgeIndex->m_Index[*i*]. The indices can be further used to find the edges' vertices through m_pEdgeVertexIndex. For example, the two vertices of the *i*-th triangle's first edge, (***v***0, ***v***1), can be visited by the following codes.

```
TTriple<SFInt32>face=m_pFaceEdgeIndex->m_Index[i];
TCouple<SFInt32>edge0=m_pEdgeVertexIndex->m_Index[face[0]];
v0=m_pCoord->m_Point[edge[0]];
v1=m_pCoord->m_Point[edge[1]];
```

The index array pointed by m_pEdgeFaceIndex records the relation between edges and triangle faces. We assume each edge in the mesh is only shared by two triangles at most. m_pEdgeFaceIndex->m_Index[*k*][0] and m_pEdgeFaceIndex->m_Index[*k*][1] are the indices of the two adjacent triangles sharing the *k*-th edge. The triangle indices can be further used to locate the vertices through m_pFaceVertexIndex.

The pointer m_pVertexFaceIndex points to an array of index array, where each index array records the relation between vertices and faces. For the *i*-th vertex, the triangle faces that shares the vertex are indexed by

```
m_pVertexFaceIndex->m_Index[i][0],
m_pVertexFaceIndex->m_Index[i][1],
…
m_pVertexFaceIndex->m_Index[i][N-1],
```

where N equals m_pVertexFaceIndex->m_Index[*i*].size().

The field m_pVertexEdgeIndex is similar to m_pVertexFaceIndex. It is an array of index array, where each index array records the indices of edges that share one vertex.

3.5.9.4 IElevationGridFeature

The IElevationGridFeature and its subclasses describe the general elevation grid and a particular type employed in terrain rendering (Fig. 3.13). The IElevationGridFeature describes all the attributes of an elevation grid except m_Height and m_YScale fields. Those two fields are described in CElevationGrid16 and CElevationGrid32 based on different height precision. Field m_Height records the height value of each vertex in the grid. Class CElevationGrid16 uses a 16-bits integer to represent height value, while class CElevationGrid32 uses a 32-bits integer. Field m_YScale describes the scale factor of height value.

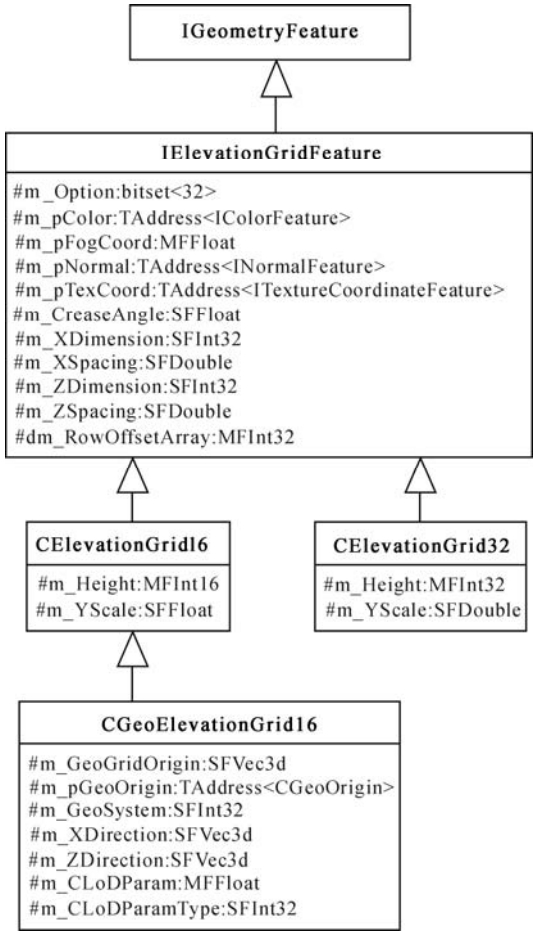

Fig. 3.13 The class hierarchy of IElevationGridFeature

For the explanation of fields m_pColor, m_pNormal, m_pTexCoord and m_CreaseAngle, please refer to the IComposedGeometryFeature section.

m_pFogCoord specifies the fog depth of each vertex, and if this field is not

NULL and fog effect is enabled, this field will replace the default depth.

m_XDimension and m_ZDimension specify the X and Z direction dimension and should be equal to each other. m_XSpacing and m_ZSpacing specify the particular distance between adjacent grid vertices in X and Z directions based on the different spatial reference frames (degrees in “GD”, meters in “UTM” and “GC”).

dm_RowOffsetArray specifies each row’s offset in m_Height field. And this array should be initialized in the height data initialization procedure. Here we use this offset array to avoid massive multiplication in the geocentric coordinate fetch procedure.

The following function shows how to get the 3D coordinate of a point in the grid:

```
SFVec3f CElevationGrid16::GetVertex(int iX, int iZ)
{
    assert (0 ≤ iX && iX < m_XDimension);
    assert (0 ≤ iZ && iZ < m_ZDimension);

    SFVec3f pos;
    pos.x = iX * m_XSpacing;
    pos.y = m_Height[iX + dm_RowOffsetArray[iZ]] * m_YScale;
    pos.z = iZ * m_ZSpacing;

    return pos;
}
```

m_Option is a 32 bits bitset. Following is the bit definition of m_Option:

Lowest order bit (the zeroth bit), CCW: If it is 1, this means that the triangle vertices are arranged in counter clock wise order. Otherwise, they are in clock wise order.

The first bit, COLORPERVERTEX: If it is 1, this means that colors are stored for each vertex. Otherwise colors are stored for each triangle. This flag bit is invalid for IElevationGridFeature, since all geometric properties in IElevationGridFeature are stored for each vertex. This flag bit is for compatibility of other classes derived from IGeometryFeature. The following shows how to fetch the color of a vertex, which is defined by GetVertex(i, j)

```
SFColor color = m_pColor->m_Color[i + dm_RowOffsetArray[j]];
```

The second bit, NORMALPERVERTEX: If it is 1, this means the normal is stored for each vertex. Otherwise, the normal is stored for each triangle. This flag bit is invalid for IElevationGridFeature since all geometric properties in IElevationGridFeature are stored for each vertex. This flag bit is for compatibility of other classes derived from IGeometryFeature.

The third bit, SOLID: If it is 1, the backface culling would be enabled. Otherwise, backface culling would be disabled and two-side lighting would be enabled.

3.5.9.5 CGeoElevationGrid16

m_GeoGridOrigin specifies the coordinate of the grid’s south-west corner. The

assignment of this field should refer to the spatial reference frame, which was defined in m_GeoSystem field.

m_GeoSystem specifies the spatial reference frame type. Currently, we have “GD”, “UTM”, “GC” and “PLANE” types defined in CGeoElevationGrid16.

“GD”: Geodetic spatial reference frame (latitude/longitude). Default coordinate order is (<latitude>, <longitude>, <elevation>). Example: (30.305773, 120.086411, 6) is the latitude/longitude coordinate for Zhejiang University, Zijingang Campus, Zhejiang, China.

“UTM”: Universal Transverse Mercator. One further required argument must be supplied for UTM in order to specify the zone number (1.60). default coordinate order is (<northing>, <easting>, <elevation>). Example: (3356259.32, 219802.63, 6) is the zone 51 northern hemisphere UTM coordinate for the same place in the above paragraph.

“GC”: Earth-fixed Geocentric with respect to the WGS84 ellipsoid. Example: (-2760454.3788, 3218501.1476, 4764637.8222) is the geocentric coordinate for the same place in the above paragraph.

“PLANE”: This mode specifies a kind of simple plane terrain and would not project to the sphere coordinate. As a lot of terrain blocks only refer to a much smaller area on the earth, it is too complex to employ the standard spatial reference frame types. Thus, we can draw a simple plane to imitate the sphere surface. This could both avoid massive computation and simplify the terrain representation.

m_pGeoOrigin specifies a local coordinate system under the particular spatial reference frame. Its variable type is CGeoOrigin and the data structure of CGeoOrigin can be referred to in Fig. 3.12. The purpose of this node is to extend the precision of vertex coordinates. Since the data of the whole earth is in a much wider range, single-precision float is insufficient. We provide this node to transform the absolute coordinate into a single-precision local coordinate system.

The Member variables of CGeoOrigin are explained as follows.

m_GeoCoords specifies the particular spatial reference frame coordinate of a local coordinate system for the terrain grid. Any subsequent geospatial locations will be translated into the absolute earth coordinate by adding m_GeoCoords.

m_GeoSystem is the same with the CGeoElevationGrid16 member variable m_GeoSystem.

The above two fields define an extended local coordinate system upon the geometry grid. In the model rendering section, we can first transform the camera into the coordinate system CGeoOrigin defined, then render the terrain grid in the local coordinate system. This operation could greatly reduce the precision lost, by adding a small offset with large geometry coordinate.

m_bRotateYUp is used to specify whether the vertex coordinates should be rotated so that their up direction is aligned with the *Y* axis. The default value of this field is FALSE. This means that the local up direction will depend upon the location of m_GeoCoords with respect to the planet surface. This is essential while performing the FLY and WALK operations upon the terrain, which assumes the up direction parallel to the *Y* axis.

m_XDirection and m_ZDirection specify the directions of *X* and *Z* axis.

m_CLoDParam and m_CLoDParamType specify the LoD attributes, which control the level details of the terrain grid. Currently, there are two kinds of LoD parameters defined in CGeoElevationGrid16. They are "VIEW_RADIUS" and "VERTEX_ERROR". "VIEW_RADIUS" specifies the viewing radius of each vertex. If the distance from view point to this vertex is less than the radius, we need to split the triangle containing this vertex. Otherwise, we will try to merge the triangle with its neighboring triangle. "VERTEX_ERROR" specifies the screen error of each vertex. If this option is specified, splitting and merging of each triangle will be determined upon the screen error of each vertex.

The following examples show how to get the vertex coordinate under different spatial reference frames.

- *Example 1*

This example shows how to get the geocentric coordinate from the "GD" elevation grid (Fig. 3.14). iX and iZ are the corresponding *X* and *Z* direction grid indices. The field values of the corresponding CGeoElevationGrid16 instance are listed in Table 3.2.

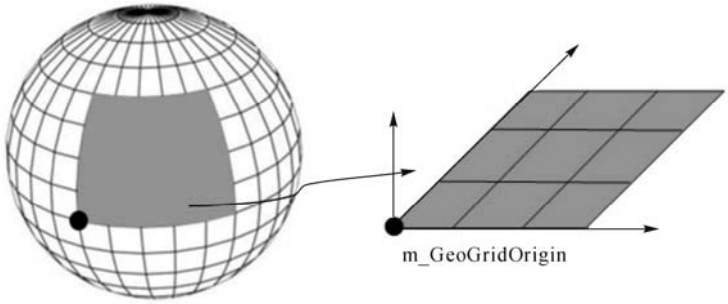

Fig. 3.14 "GD" spatial reference frame without m_GeoOrigin

Table 3.2 The field values of the CGeoElevationGrid16 instance

CGeoElevationGrid16 grid	Field value
…	…
grid.m_Height	…
grid.m_GeoGridOrigin	(30.305773, 120.086411, 6)
grid.m_GeoSystem	"GD"
grid.m_pGeoOrigin	NULL
grid.m_CLoDParam	…
grid.m_CLoDParamType	…

The following pseudo-code fragment simply shows how to fetch a 3D coordinate under "GD" spatial reference frame:

```
// get local "GD" coordinate
SFVec3d posSRF;
posSRF.x = iX*grid.m_XSpacing*PI / 180.0f;
```

```
posSRF.y = grid.m_Height[iX + grid.dm_RowOffsetArray[iZ]]*
grid.m_YScale;
posSRF.x = iX*grid.m_XSpacing*PI/180.0f;

// get global "GD" coordinate
SFVec3d gbPosSRF;
gbPosSRF = grid.m_GeoGridOrigin + posSRF;

double R = EarthRadius + gbPosSRF.y;

// transform "GD" coordinate into "GC" coordinate
SFVec3d gcCoord;
gcCoord.x = R*cos(gbPosSRF.z)*cos(gbPosSRF.x);
gcCoord.y = R*sin(gbPosSRF.x);
gcCoord.z = R*sin(gbPosSRF.z)*cos(gbPosSRF.x);

return gcCoord;
```

- *Example 2*

The following example shows how to get the vertex coordinate from a CGeoElevationGrid16 instance (Fig. 3.15), which has m_pGeoOrigin not NULL. The field values of the corresponding CGeoElevationGrid16 instance are listed in Table 3.3.

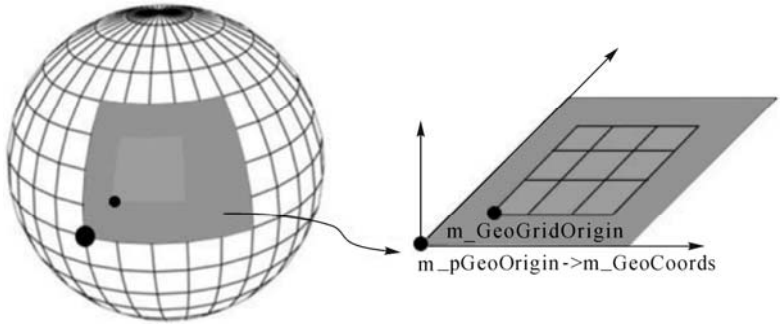

Fig. 3.15 "GC" grid mixed with "GD" geoOrigin

Table 3.3 The field values of the CGeoElevationGrid16 instance

CGeoElevationGrid16 grid	Field value
…	…
grid.m_Height	…
grid.m_GeoGridOrigin	(1346023.14, 4642195.79, 4025252.11)
grid.m_GeoSystem	"GC"
grid.m_pGeoOrigin->m_GeoCoords	(30.1234e, 45.2345n, 0)
grid.m_pGeoOrigin->m_GeoSystem	"GD"
grid.m_pGeoOrigin->m_bRotateYUp	FALSE
grid.m_CLoDParam	…
grid.m_CLoDParamType	…

This example shows the combination of "GD" and "GC" spatial reference frames. Type of terrain grid is "GC" and "GD" for extended local coordinate system. The following is the pseudo-code fragment.

```
// get the local coordinate of a vertex under "GC" mode
SFVec3d gcGridCoord;

lcGridCoord.x = grid.m_GeoGridOrigin.x+iX*grid.m_XSpacing;
lcGridCoord.y = grid.m_GeoGridOrigin.y+
    m_Height[iX+dm_RowOffsetArray[iZ]]*m_YScale;
lcGridCoord.z = grid.m_GeoGridOrigin.z+iZ*grid.m_ZSpacing;

// transform GeoOrigin's coordinate into "GC" coordinate SFVec3d
gcGeoOriginCoord;

// transform coordinate of m_GeoOrigin, which was in the "GD"
// spatial reference frame, into geocentric coordinate
// "GD" to "GC" formula
// x = Rcos(latitude)cos(longitude);
// y = Rsin(latitude);
// z = Rcos(latitude)sin(longitude);
gcGeoOriginCoord.x = TransformCoord("GC",
    grid.m_GeoOrigin->m_GeoSystem,
    grid.m_GeoOrigin->m_GeoCoords);

//get the absolute coordinate by adding GeoOrigin with local Coordinate
return gcGeoOriginCoord+gcGridCoord;
```

- *Example 3*

This example shows the UTM spatial reference frame (Fig. 3.16). We omit the detailed transformation steps from UTM to geodetic coordinates, since it is quite complicated. The field values of the corresponding CGeoElevationGrid16 instance are listed in Table 3.4.

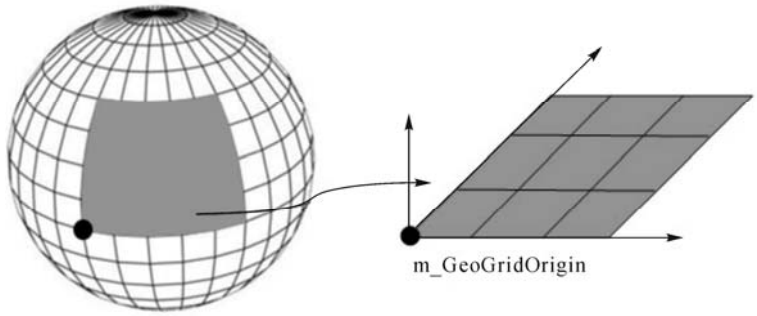

Fig. 3.16 "UTM" spatial reference frame schema

Table 3.4 The field values of the CGeoElevationGrid16 instance

CGeoElevationGrid16 grid	Field value
…	…
grid.m_Height	…
grid.m_GeoGridOrigin	(3356259.32, 219802.63, 6)
grid.m_GeoSystem	"UTM"
grid.m_pGeoOrigin	NULL
grid.m_CLoDParam	…
grid.m_CLoDParamType	…

The following pseudo-code shows how to fetch the coordinate under “UTM” and transform it into “GD” coordinate:

```
// get local "UTM" coordinate
SFVec3d posSRF;
posSRF.x = iX*grid.m_XSpacing;
posSRF.y = grid.m_Height[iX+grid.dm_RowOffsetArray[iZ]]
* grid.m_YScale;
posSRF.x = iX * grid.m_XSpacing;

// get global "GD" coordinate
SFVec3d gbPosSRF;
gbPosSRF = grid.m_GeoGridOrigin + posSRF;

// transform "UTM" coordinate into "GD" coordinate and return
// For detailed algorithm, please refer to Universal Transverse
// Mercator Grid (Department of Army, 1973)
return TransformUTM2GD(SFVec3d pos);
```

3.5.9.6 ICurveFeature

ICurveFeaure is the base class to represent the features of all kinds of 3D curves. The geometric features represented by its concrete subclasses are mainly the same as their name implies. CPoint is also derived from ICurveFeature for convenience.

CMLine represents multiple independent line segments. CPolyline represents a curve composed of multiple connected line segments. CBSpline represents a B-Spline curve.

The concrete data members of these classes are not listed here to save space.

3.5.9.7 ITextFeature

ITextFeature represents the geometry for 3D texts. CText is the concrete class of ITextFeature. CMText is the compact representation of multiple CText classes.

3.5.10 IAppearanceFeature and Related Features

IAppearanceFeature and its subclasses represent the appearance of shape. The related class diagram is shown in Fig. 3.17. IAppearanceFeature is an abstract class, and has no data member. CAppearance is one of the concrete classes of IAppearanceFeature. It has several shaders, one material, one texture and one texture transform. So far, CAppearance is the most important appearance feature in Visionix.

The types of all data members of IAppearanceFeature subclasses are derived from IAppearanceChildFeature. There are four abstract classes derived from IAppearanceChildFeature, which are IMaterialFeature, ITextureFeature,

ITextureTransformFeature and IShaderFeature. They respectively describe the surface material, texture, texture transform and shader of the shape. The class diagram related to IAppearanceChildFeature is shown in Fig. 3.18.

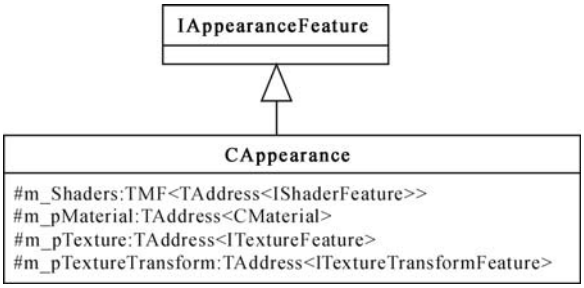

Fig. 3.17 Class IAppearanceFeature and CAppearance

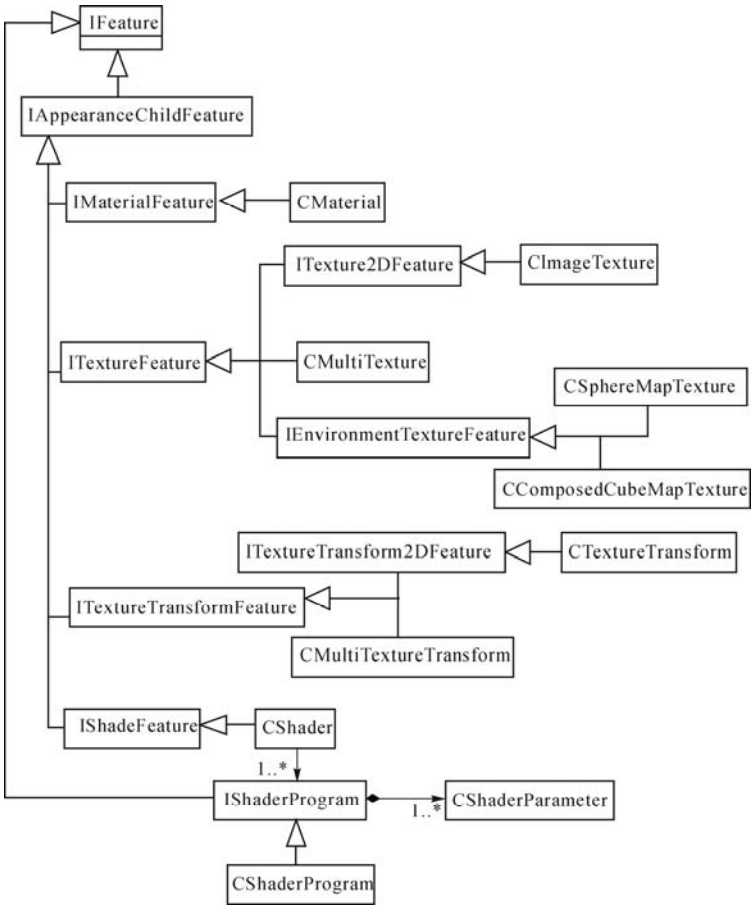

Fig. 3.18 The subclasses of IAppearanceChildFeature

3.5.10.1 IMaterialFeature

IMaterialFeature is the abstract class which describes lighting and materials. CMaterial is the most important subclass.

```
class CMaterial:public class IMaterialFeature
{
protected:
    SFFloat m_AlphaCullingThreshold;
    SFFloat m_Transparency;
    SFFloat m_Shininess;
    SFFloat m_Reflectance;
    SFFloat m_AmbientIntensity;
    SFColorRGB m_EmissiveColor;
    SFColorRGB m_DiffuseColor;
    SFColorRGB m_SpecularColor;
    SFByte m_ShadingMode;
    // The member functions are omitted.
}
```

The member variables in CMaterial define the material parameters which are used in lighting computation. The light intensity at the point $\boldsymbol{p}$ on the object's surface is computed by the following pseudo-codes.

```
if alpha < m_AlphaCullingThreshold
    I(p) = I_background(p)
else
    I(p) =m_AmbientIntensity*Light_ambient + m_DiffuseColor*Light*(N*L)
         + m_SpecularColor*Light*(N*H)^m_Shininess
         + m_Transparency*I_refraction(p)
         + m_Reflectance*I_reflectance(p)
```

The variable alpha is the opacity value at point $\boldsymbol{p}$ on the surface. When alpha is less than the given threshold m_AlphaCullingThreshold, $\boldsymbol{p}$ is treated as it is totally transparent. The brightness at point $\boldsymbol{p}$ is equal to the background's light intensity.

When alpha is larger than the threshold, the light intensity at point $\boldsymbol{p}$ is the sum of local illumination at $\boldsymbol{p}$ and the global reflected and refracted illumination. The local illumination at point $\boldsymbol{p}$ is computed by Phong lighting model. Strictly speaking, we need to adopt the global illumination model to compute $I(\boldsymbol{p})$. However, some approximation is adopted in real applications, e.g., $I_{\text{refraction}}(\boldsymbol{p})$ is approximated by $I_{\text{background}}(\boldsymbol{p})$ and $I_{\text{reflectance}}(\boldsymbol{p})$ is computed by environment mapping. The rendering module in the rendering system determines the adopted computation method.

Each object in Visionix owns its material. If no specific material is assigned, the system assigns the default material of diffuse color to the object.

3.5.10.2 ITextureFeature

ITextureFeature is an abstract class for describing textures. There are three subclasses derived from it:

- ITexture2DFeature;
- IEnvrionmentTextureFeature;
- CMultiTexture.

ITexture2DFeature is an abstract class for describing two-dimensional or one-dimensional textures.

```
class ITexture2DFeature:public ITextureFeature
{
public:
    enum {LOOP, REPEATS, REPEATT, TEX2DFEATURE_MUSK_NUM=32 };
protected:
    std::bitset<32> m_Option;
    SFByte m_MinFilterType;
    SFByte m_MagFilterType;
}
```

m_Option is used to define some texture attributes: (1) The first bit is only used for movie texture, which indicates whether the movie is to be played repeatedly; (2) The second bit is to indicate whether the texture coordinates at *S*-axis direction adopt the wrap mode; (3) The third bit is to indicate whether the texture coordinates at T-axis direction adopt the wrap mode.

m_MinFilterType and m_MagFilterType respectively represent the texture filters used in texture magnification and minification.

3.5.10.3 CImageTexture

The class CImageTexture represents the most common texture type which uses one image to define a texture. CImageTexture contains the URLs which point to the image (URL is written as MFString to be compatible with X3D standard, which indicates that multiple URLs can provide the texture image) and the real image data TAddress<IImageFeature> m_pImage which is defined as follows:

```
class CImageTexture:public ITexture2DFeature
{
  protected:
    TAddress<IImageFeature>     m_pImage;
    MFString                    m_URL;
};
```

IImageFeature is the abstract class of all image types in Visionix. It is defined as follows:

```
class IImageFeature:public IFeature
{
  protected:
    SFUInt32 m_Width;
    SFUInt32 m_Height;
    SFByte m_Components;
    SFByte m_ImageType;
    SFByte m_CompressionType;
    SFByte m_PyramidLevelCount;
    SFUInt32 m_DataSize;
    SFByte* m_pData;
};

class CImage:public IImageFeature
{};
```

CImage is a subclass of IImageFeature. It is actually the implementation class of IImageFeature (with no new data members introduced).

The data members of m_Width, m_Height and m_Components are respectively used to describe the height, width and color channels of the image. The fields of m_DataSize and m_pData are used to describe the actual data size in the buffer and the pointer to the buffer.

The data member of m_ImageType is used to describe the data format in the image buffer. It can be of any value of the following enumeration type Enum_ImageType:

```
enum Enum_ImageType
{
        IMAGE_TYPE_RAW = 0,
        IMAGE_TYPE_FILE_PACKAGE_JPG,
        IMAGE_TYPE_FILE_PACKAGE_TGA,
        IMAGE_TYPE_FILE_PACKAGE_PNG,
        IMAGE_TYPE_FILE_PACKAGE_BMP,
        IMAGE_TYPE_FILE_PACKAGE_TIFF,
        IMAGE_TYPE_RAW_HDI_SHORT,
        IMAGE_TYPE_RAW_HDI_UNSIGNED_SHORT,
        IMAGE_TYPE_RAW_HDI_INT,
        IMAGE_TYPE_RAW_HDI_UNSIGNED_INT,
        IMAGE_TYPE_RAW_HDI_HALF_FLOAT,
        IMAGE_TYPE_RAW_HDI_FLOAT,
        IMAGE_TYPE_RAW_HDI_DOUBLE,
};
```

Such enumeration values have the following meanings:

IMAGE_TYPE_RAW: This indicates the pixel data are preserved in uncompressed or compressed raw format. It needs to be combined with m_CompressionType to

judge how the pixel data is preserved. The so-called uncompressed raw data means that the data are saved in the buffer from top to bottom and from left to right according to line priority order. The number of each pixel's color compoment is determined by m_Components. The size of each compoment data is 8 bits long. The compressed raw format is explained in m_CompressionType.

IMAGE_TYPE_FILE_PACKAGE_*X*: The enumeration value prefixed by IMAGE_TYPE_FILE_PACKAGE_ indicates that the buffer stores the memory image of the file. The last identifier *X* represents the file type, i.e. JPG, TGA, PNG, BMP, and TIFF shows that the file type is jpg, tga, png, bmp and tiff.

IMAGE_TYPE_RAW_HDI_*Y*: The enumeration value prefixed by IMAGE_TYPE_RAW_HDI_*Y* indicates that the file saved in raw format is a high range image. The last identifier *Y* represents the data type of each color compoment, e.g., SHORT means each color component is represented by a short integer of 16 bits.

The data member m_CompressionType in IImageFeature indicates how the image data in the buffer is compressed. This field is valid only when m_ImageType is IMAGE_TYPE_RAW.

```
enum Enum_CompressionType
{
    NO_COMPRESSION = 0,
    RGB_S3TC_DXT1_COMPRESSION,
    RGBA_S3TC_DXT1_COMPRESSION,
    RGBA_S3TC_DXT3_COMPRESSION,
    RGBA_S3TC_DXT5_COMPRESSION,
};
```

NO_COMPRESSION means that the data in the buffer is in uncompressed raw format. The compressed raw format is the data format produced by compressing the uncompressed raw format data.

So far a lot of 3D display cards support DXT (also called S3TC) image compression format. So we adopt S3TC compression format in Visionix to compress data. We provide RGB_S3TC_DXT1_COMPRESSION compression scheme for RGB image whose m_Component is 3. The other three compression schemes RGBA_S3TC_DXT1_COMPRESSION, RGBA_S3TC_DXT3_COMPRESSION, RGBA_S3TC_DXT5_COMPRESSION are provided for RGBA image whose m_Component is 4. The above three compression schemes differ in their compression ratios.

3.5.10.4 IEnvironmentTextureFeature

IEnvironmentTextureFeature is the texture abstract class for environment mapping. Its subclasses CSphereMapTexture and CComposedCubeMapTexture respectively correspond to sphere environment mapping and cube environment mapping. Environment mapping is always applied to simulate reflection approximately.

Sphere environment mapping is equivalent to putting a perfectly smooth ball

at the center of the scene and taking pictures of this ball from very distant locations with a telephoto lens. The class CSphereMapTexture can use one texture (pointed to by m_pTexture) to perform sphere environment mapping to save memory and improve efficiency in some applications.

```
class CSphereMapTexture:public IEnvironmentTextureFeature
{
protected:
   TAddress<ITextureFeature> m_pTexture;
}
```

The cube texture, represented by the class CComposedCubeMapTexture, is a special technique which uses six 2D texture images to build a textured cube centered at the origin. Each data member represents one texture of the six textures. The declaration of CComposedCubeMapTexture is as follows:

```
Class CComposedCubeMapTexture:public IEnvironmentTextureFeature
{
   TAddress<ITextureFeature> m_pFrontTexture;
   TAddress<ITextureFeature> m_pBackTexture;
   TAddress<ITextureFeature> m_pLeftTexture;
   TAddress<ITextureFeature> m_pRightTexture;
   TAddress<ITextureFeature> m_pTopTexture;
   TAddress<ITextureFeature> m_pBottomTexture;
}
```

3.5.10.5 CMultiTexture

The class CMultiTexture represents multi-layer textures on the object's surface and defines how these textures blend together. The data members are defined as follows:

```
class CMultiTexture:public ITextureFeature
{
Protected:
   TMF<TAddress<ITextureFeature>>m_Textures;
   SFColorRGB    m_Color;
   SFFloat       m_Alpha;
   MFByte        m_Mode;
   MFByte        m_Usage;
   MFByte        m_Source;
   MFFloat       m_Coefficients;
   MFByte        m_Function;
}
```

Each TAddress<ITextureFeature> in the array of m_Textures points to an instance of ITextureFeature which represents one layer of texture. Note that it is forbidden in Visionix that TAddress<ITextureFeature> points to another CMultiTexture instance, which means that multi-textures are forbidden to be nested.

Before we explain the data member of multi-textures, we first explain how we implement CMultiTexture by utilizing multiple textures extensions in OpenGL in Visionix. Actually we tailor the function of multiple textures in OpenGL. The data flow is shown in Fig. 3.19.

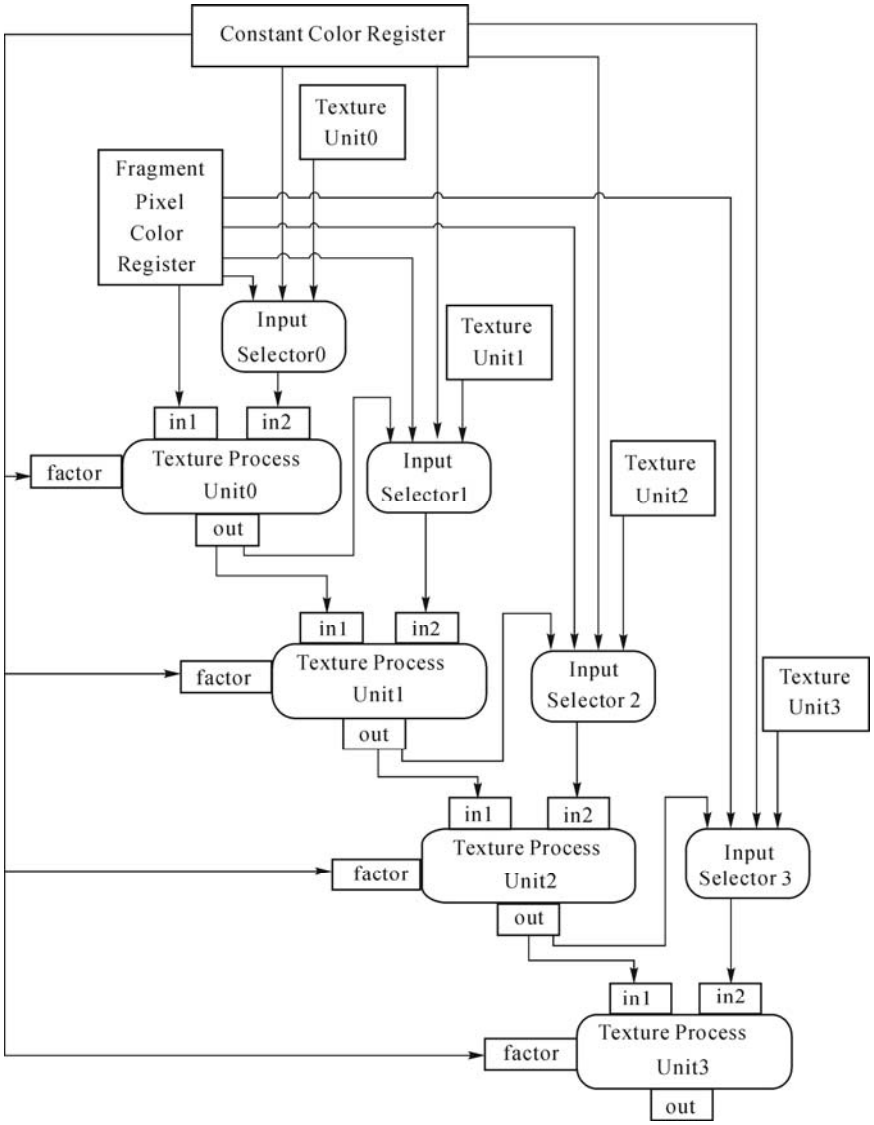

Fig. 3.19 The data flow for multiple texture mapping

The Texture Process Unit (TPU) in the figure is used to combine two inputs (from in1, in2) of pixel/texel colors into one new color and output the new color via the out port. Note here the colors are all represented in RGBA format. Input Selector selects one input from multiple inputs. Each Texture Unit (TU) is used to store one layer of texture, which outputs texel color according to texture coordinates.

In Fig. 3.19 four TPUs are displayed (which means four layers of texture mappings can be realized). The input in1 of TPU0 is fixed—the Fragment Pixel Color Register (FPCR). This register stores the color of any point $\boldsymbol{p}$ of the rasterized fragments of the faces to be rendered. For the point $\boldsymbol{p}$, the process of multiple texture mapping is actually to synthesize multiple textures on the base of p's color.

The input in2 of TPU0 is selected by Input Selector0. Input Selector0 has three inputs: Constant Color Register (CCR), Texture Unit0 and FPCR. CCR stores a color constant. Texture Unit0 outputs the texel color of the saved textures. TPU0 combines the inputs from in1 and in2 in some way. During the combination, CCR will be transferred to TPU0 as a factor (but whether to adopt it or not depends on the combination way). The combination color is output via the out port.

TPU1 differs from TPU0 a bit. The input in1 of TPU1 is the output of TPU0. The input in2 of TUP1 is the output of Input Select1. Input Select1 has four inputs, which are TPU0.out, Constant Color Register, TU1 and FPCR. TPU2 and TPU3 are similar to TPU1. The final four-layer texture mapping result is output via TPU3's out port.

If we understand how multiple texture mapping is realized, it is easier to understand CMultiTexture.

How many layers of textures can be stored in CMultiTexture is determined by the number of TPUs. All arrays in CMultiTexture have the same length. In actual implementation, the largest number of TUP is determined by the graphics hardware (the corresponding value is Texture Stage in OpenGL).

The data members m_Color and m_Alpha represent color and alpha stored in Constant Color Register respectively. The data member m_Mode is to control the combination fashion of TPU. The data member m_Source is to control how Input Select makes selection. The data member m_Usage is used to describe the texture usage. The data member m_Coefficients and m_Function are reserved fields.

m_Mode is an array of bytes, where each element is a enumeration value of Enum_Mode type. m_Mode[i] indicates how to combine a new color from two input colors for TPU_i.

```
enum Enum_Mode{
    MODE_MODULATE,
    MODE_REPLACE,
    MODE_ADD,
    MODE_SUBTRACT,
    MODE_ADDSMOOTH,
    MODE_BLENDDIFFUSEALPHA,
    MODE_BLENDTEXTUREALPHA,
    MODE_BLENDFACTORALPHA
};
```

The implications are shown in Table 3.5.

Table 3.5 Mode type

Mode	Meaning
MODE_MODULATE	out_i.RGBA= in1.RGBA*in2.RGBA
MODE_REPLACE	out_i.RGBA =in2.RGBA
MODE_ADD	out_i.RGBA =in1.RGBA+in2.RGBA
MODE_SUBTRACT	out_i.RGBA =in1.RGBA-in2.RGBA
MODE_ADDSMOOTH	out_i.RGBA =in1.RGBA+in2.RGBA-in1.RGBA*in2.RGBA =in1.RGBA+(RGBA(1,1,1,1) - in1.RGBA) * in2.RGBA
MODE_BLENDDIFFUSEALPHA	out_i.RGBA=in1.RGBA*(FPCR.alpha)+in2.RGBA*(1-FPCR.alpha), FPCR.alpha is the alpha value in the Fragment Pixel Color Register
MODE_BLENDTEXTUREALPHA	out_i.RGBA=in1.RGBA*($texture_i$.alpha)+in2.RGBA * (1–$texture_i$.alpha), $texture_i$.alpha is the alpha value of the corresponding texel in the i-th Texture Unit
MODE_BLENDFACTORALPHA	out_i.RGBA=in1.RGBA*(CCR.alpha)+in2.RGBA * (1–CCR.alpha), CCR.alpha is the alpha value in the Constant Color Register

The data member m_Source is also an array of bytes, where each element is the enumeration value of Enum_Source type. m_Source[*i*] indicates which input the *i*-th Input Select should select.

```
enum Enum_Source{
      SOURCE_DEFAULT,
       SOURCE_PREVIOUS,
       SOURCE_FACTOR,
       SOURCE_PRIMARY
};
```

The meaning of these enumeration values are shown in Table 3.6.

Table 3.6 Source type

Source	Description
SOURCE_DEFAULT	TPU_i.in2 = the texel in the *i*-th TextureUnit = m_Textures[*i*]
SOURCE_PREVIOUS	TPU_i.in2 = TPU_{i-1}.out
SOURCE_FACTOR	TPU_i.in2 = CCR.RGBA = (m_Color, m_Alpha)
SOURCE_PRIMARY	TPU_i.in2 = FPCR.RGBA

Different from m_Mode and m_Source, m_Usage is not for control use during multiple texture mapping. It is only descriptive. m_Usage[*i*] indicates the meaning of m_Textures[*i*]. Each element in m_Usage can have one of the following values.

```
enum Enum_Usage{
      USAGE_DIFFUSE_MAP = 0,
      USAGE_LIGHT_MAP,
      USAGE_REFLECTION_MAP,
      USAGE_NORMAL_MAP,
      USAGE_SPECULAR_MAP
};
```

The meaning of these enumeration values are shown in Table 3.7.

Table 3.7 Usage type

Usage	Description
USAGE_DIFFUSE_MAP	Texture for diffuse component
USAGE_LIGHT_MAP	Texture for radiance
USAGE_REFLECTION_MAP	Texture for reflection
USAGE_NORMAL_MAP	Texture for normal vectors
USAGE_SPECULAR_MAP	Texture for specular component

The following gives some examples of multiple texture mapping. Suppose A, B and C are three textures, and m_Alpha is a floating point number between 0 and 1.

(1) To realize the effect of A*B*C, set a CMultiTexture instance as follows:

```
m_Textures[]={A,B,C};
m_mode[]={MODE_REPLACE,MODE_MODULATE,MODE_MODULATE};
m_Source[]={SOURCE_DEFAULT, SOURCE_DEFAULT,SOURCE_DEFAULT};
m_Color=SFColorRGB(1.0,1.0,1.0);
m_Alpha=1.0;
```

(2) To realize the effect of A*B+C, set a CMultiTexture instance as follows:

```
m_Textures[]={A,B,C};
m_mode[]={MODE_REPLACE,MODE_MODULATE,MODE_ADD};
m_Source[]={SOURCE_DEFAULT,SOURCE_DEFAULT,SOURCE_DEFAULT};
m_Color=SFColorRGB(1.0,1.0,1.0);
m_Alpha=1.0;
```

(3) To realize the effect of A*B*a+(1-a)*C, set a CMultiTexture instance as follows:

```
m_Textures[]={A,B,C};
m_mode[]={MODE_REPLACE,
          MODE_MODULATE,MODE_BLENDFACTORALPHA};
m_Source[]={SOURCE_DEFAULT,SOURCE_DEFAULT,SOURCE_DEFAULT};
m_Color=SFColorRGB(1.0,1.0,1.0);
m_Alpha=a;
```

If merely counting on OpenGL's fixed rendering pipeline, the combination fashions for multiple textures are quite limited and simple. Visionix can realize more complex combination fashions through programmable shaders.

3.5.10.6 ITextureTransformFeature

ITextureTransformFeature is the abstract class to describe texture transformation features. It has no data members defined. CTextureTransform realizes the texture transformation for one-layer of texture. CMultiTextureTransform realizes the texture transformation for multiple layers of textures.

3.5.10.7 CTextureTransform

The declaration of CTextureTransform is listed as follows:

```
class CTextureTransform:public ITextureTransform2DFeature
{
    mutable std::bitset<32>dm_Option;
    mutable BasicType::SFMatrix44f dm_Matrix;
    SFFloat m_Rotation;
    SFVec2f m_Center;
    SFVec2f m_Scale;
    SFVec2f m_Translation;
}
```

Suppose the texture coordinate is defined in the s-t coordinate frame, and the transformation is represented by a matrix. CTextureTransform actually defines the following transformation,

$$(\boldsymbol{s}',\boldsymbol{t}')^{\boldsymbol{T}} = M_C(\text{-m\_Scale})\times M_S(\text{m\_Scale})\times M_R(\text{m\_Rotation}) \times M_T(\text{m\_Center})\times M_T(\text{m\_Translation})\times(\boldsymbol{s},\boldsymbol{t})^{\boldsymbol{T}} \quad (3.1)$$

where,

(1) $(\boldsymbol{s},\boldsymbol{t})^{\boldsymbol{T}}$ is the texture coordinate before transformation and $(\boldsymbol{s}',\boldsymbol{t}')^{\boldsymbol{T}}$ is the texture coordinate after transformation.

(2) M_T(m_Translation), M_T(m_Center) and M_T(-m_Center) respectively represent the translation transformations with translation vectors m_Translation, m_Center and -m_Center.

(3) M_R(m_Rotation) represents the counter-clockwise rotation and the rotation radian is m_Rotation.

(4) M_S(m_Scale) represents the scaling transformation. The scale value along the *s*-axis is m_Scale.v[0] and the scale value along the *t*-axis is m_Scale.v[1].

Note that the transformation here is in the opposite order to the transformation defined in SFTransform3{f|d}. The reason is that the transformations are considered

to be performed on texture coordinates instead of texture elements. If we consider the effect on the surface of the geometry, the effects of these operations are just opposite. For example, the scale value (2,2) to enlarge texture coordinates actually has the effect of shrinking the texture's size to half of its original size.

3.5.10.8 CMultiTextureTransform

```
class CMultiTextureTransform:public ITextureTransformFeature
{
protected:
    TMF<TAddress<ITextureTransformFeature> > m_TextureTransform;
}
```

The class CMultiTextureTransform should be used together with CMultiTexture. It defines the array m_TextureTransform where each element points to an ITextureTransformFeature instance. m_TextureTransform[*i*] provides the texture coordinate transformation for the corresponding m_Textures[*i*] in CMultiTexture.

If we use a CMultiTexture instance to represent the texture feature, but not use CMultiTextureTransform, then the texture mapping is disabled. If a CMultiTextureTransform instance has fewer transformations than the number of texture layers, the last texture transform in the CMultiTextureTransform instance would be used to provide the coordinate transformation for the remaining textures.

3.5.10.9 IShaderFeature

The class IShaderFeature describes programmable GPU shaders in Visionix. With shaders we can directly operate vertices and fragments in the graphics pipeline. The base class IShaderFeature does not include any concrete implementations. It is defined as follows:

```
class IShaderFeature:public IAppearanceChildFeature
{
  public:
     enum Enum_Language
     {
        LANGUAGE_CG=0,
        LANGUAGE_GLSL,
        LANGUAGE_HLSL
     };
  protected:
     SFBool m_bActiveFlag;
     SFByte m_Language;
}
```

The class IShaderFeature has the data members m_bActiveFlag and m_Language, which are used to respectively describe whether the shader is used and which

language is adopted. Three kinds of GPU programmable languages, which are nVIDIA Cg, GLSL and HLSL can be specified. Cg supports both OpenGL and DirectX, GLSL is only supported by OpenGL, and HLSL is supported by DirectX. The current vision Visionix only supports Cg.

3.5.10.10 CShader

The class CShader is the concrete subclass of IShaderFeature. It is defined as follows:

```
class CShader:public IShaderFeature
{
      TMF<TAddress<IShaderProgram>> m_ShaderPrograms;
};

typedef CShaderParameter SFNamedVariant;
class IShaderProgram:public IFeature
{
protected:
      TMF<SFString> m_SystemVariables;
      TMF<SFNamedVariant> m_UserDefinedConstants;
};

class CShaderProgram:public IShaderProgram
{
public:
      enum Enum_ProgramType
      {
            PRGTYPE_VERTEX = 0,
            PRGTYPE_FRAGMENT
            PRGTYPE_GEOMETRY
      };
Protected:
      MFString      m_Profiles;
      SFByte        m_ProgramType;
      SFString      m_Program;
      MFString      m_URL;
}
```

A CShader instance has multiple CShaderProgram instances and each CShaderProgram instance defines a shader.

The data member m_Profiles describes the profiles which the shader can use under different graphics hardware configurations through multiple strings. The following is an example.

m_Profiles[0] = “NV_GF8_8800, NV_GF9_9800: VP20”;

m_Profiles[1] = “NV_GF200_290+: VP30+”;

From the above examples, each string is divided into two parts which are separated by a colon. The first part includes the hardware information which is separated by the comma. The format of the hardware information is: vendor_

model_number. NV_GF_8800 represents the nVidia GeForce 8800. NV_GF200_290+ represents the GTX series 290 and higher versions. The second part includes the profiles for vertex program (VP), fragment program (FP) and geometry program (GP).

The data member m_ProgramType indicates the type of the shader program, which can be any enumeration value of Enum_ProgramType. The enumeration types respectively correspond to the vertex shader program and fragment shader program and geometry shader program.

The data member m_Program stores the shader program in strings.

The data member m_URL indicates the file name of the shader program. It can have multiple file names, which means that this program has multiple file copies. When m_Program is not empty, the codes are stored in m_Program. When m_Program is empty, the codes are stored in the file pointed by m_URL.

The data member m_SystemVariables defines an array of names of predefined system variables in Visionix which the shader program can visit. The data member m_UserDefinedConstants defines the constants used in the shader programs.

If a shader program variable $\boldsymbol{x}_{\text{shader}}$ is declared with the same name as that of the corresponding predefined system variable $\boldsymbol{x}_{\text{visionix}}$, the shader program can obtain the value of $\boldsymbol{x}_{\text{visionix}}$ through $\boldsymbol{x}_{\text{shader}}$ during runtime. There is a scheme to realize such a kind of binding between $\boldsymbol{x}_{\text{shader}}$ and $\boldsymbol{x}_{\text{visionix}}$ in Visionix.

The commonly-used system variables are defined as Table 3.8.

Table 3.8 The commonly-used system variables

System variable	Data Type in Visionix	Data Type in nVIDIA Cg
System_ModelMatrix	SFMatrix44f	float4x4
System_ViewMatrix	SFMatrix44f	float4x4
System_ProjMatrix	SFMatrix44f	float4x4
System_ModelViewProj	SFMatrix44f	float4x4
System_EyePosition	SFVec3f	float3
System_EyeTarget	SFVec3f	float3
System_EyeUp	SFVec3f	float3
System_Lights_Position	MFVec3f	float3 *
System_Lights_Count	SFInt	Int
Light__name_Position	SFVec3f	float3
Shape_DiffuseMap	SFImage	sampler2D
Shape_LightMap	SFImage	sampler2D
Shape_NormalMap	SFImage	sampler2D
Shape_SpecularMap	SFImage	sampler2D
Shape_ReflectionMap	SFImage	sampler2D
Shape_Texture*i*	SFImage	sampler2D
Shape_DiffuseColor	SFColorRGB	float3
Shape_EmissiveColor	SFColorRGB	float3
Shape_SpecularColor	SFColorRGB	float3
Shape_AmbientIntensity	SFFloat	float
Shape_Shininess	SFFloat	float
Shape_Transparency	SFFloat	float
Shape_Reflectance	SFFloat	float

The system variables prefixed by “System” describe some global states when the shader programs execute.

(1) System_ModelMatrix: The transformation from the local coordinate frame of the current to-be-rendered feature to the world coordinate frame.

(2) System_ViewMatrix: The transformation from the world coordinate frame to the viewpoint coordinate frame of current camera.

(3) System_ProjMatrix: The projection matrix from the viewpoint coordinate frame to the viewport coordinate frame.

(4) System_ModelViewProj: The transformation from the local coordinate frame to the viewpoint coordinate frame.

(5) System_EyePosition, System_EyeTarget and System_EyeUp: The position, target and up direction of current viewpoint.

(6) System_Lights_Position, System_Lights_Count: The positions and numbers of all lights in the scene.

(7) Light__name_Position: The position of the ILightFeature instance with the name of “name”. We usually name a feature with the format of (Feature Type) +”__”+ Name + “_”, e.g. Light__RoomLight0_Position represents the position of a light with the name RoomLight0.

(8) Shape_*X*: The attribute *X* of the relevant IShapeFeature instance (the IShapeFeature instance related to the shader program). Shape__name_*X* represents the attribute *X* of an IShapeFeature instance with the name of “name”. We illustrate *X* further as follows:

- DiffuseMap, LightMap, NormalMap, SpecularMap and ReflectionMap respectively represent the texture layer of a CMultiTexture instance with the m_Usage being USAGE_DIFFUSE_MAP, USAGE_LIGHT_MAP, USAGE_NORMAL_MAP, USAGE_SPECULAR_MAP and USAGE_REFLECTION_MAP.
- Texture*i* represents the *i*-th texture layer in the multiple texture object.
- DiffuseColor, EmissiveColor, SpecularColor, AmbientIntensity, Shininess, Transparency, Reflectance respectively correspond to the material attributes.

3.5.10.11 Appearance Modeling Examples

We list some examples below to explain how to use the above appearance features to model various appearances.

(1) Illuminated by local illumination model with diffuse texture.

The rendering equation of the point $\boldsymbol{p}$ on the surface with this material is:

$$I(\boldsymbol{p}) = \sum_i L_i k_{\text{diffuse}}(\boldsymbol{p}) \cos\left(\theta_i(\boldsymbol{p})\right) = k_{\text{diffuse}}(\boldsymbol{p}) \sum_i L_i \cos\left(\theta_i(\boldsymbol{p})\right) \tag{3.2}$$

L_i denotes the *i*-th light source’s intensity. $\theta_i(\boldsymbol{p})$ is the angle between the normal at $\boldsymbol{p}$ and the vector from $\boldsymbol{p}$ to the *i*-th light source. $k_{\text{diffuse}}(\boldsymbol{p})$ is the diffuse coefficient at $\boldsymbol{p}$, and varies with $\boldsymbol{p}$’s locations on the surface. We store the diffuse

coefficients as a texture to build the correspondence between $\boldsymbol{p}$ and the diffuse coefficient.

In order to render geometries with the illuminated results of this material, the following settings are adopted. We assume the shape with this material is denoted by s and its geometric type is of CIndexTriangleSet.

s.m_pGeometry->m_pColor->m_Color= $\{\sum_i L_i \cos(\theta_i(\boldsymbol{p}_k)) \mid \boldsymbol{p}_k$ is the k-th vertex.$\}$.

We need to pre-compute the following values:

* (s.m_pAppearance->m_pTexture) = M, which stores the diffuse coefficient texture
* (s.m_pGeometry->m_pTexCoord) = $\{\boldsymbol{u}_k \mid \boldsymbol{u}_k$ is the texture coordinate of vertex $\boldsymbol{p}_k$, which satisfies $M(\boldsymbol{u}_k) = k_{\text{diffuse}}(\boldsymbol{p}_k)\}$

When we use a single texture and m_pColor is not empty, the texture blending mode is automatically set to be "MODE_MODULATE".

(2) Inter-reflection considered with diffuse texture.

The rendering equation of the point $\boldsymbol{p}$ on the surface with this material is:

$$I(\boldsymbol{p}) = \int_\Omega L(\boldsymbol{\omega}) \cdot k_{\text{diffuse}}(\boldsymbol{p}) \cdot (-\boldsymbol{\omega} \cdot \boldsymbol{n}) \mathrm{d}\boldsymbol{\omega} = k_{\text{diffuse}}(\boldsymbol{p}) \cdot \int_\Omega L(\boldsymbol{\omega}) \cdot (\boldsymbol{p}) \cdot (-\boldsymbol{\omega} \cdot \boldsymbol{n}) \mathrm{d}\boldsymbol{\omega} \quad (3.3)$$

$\int_\Omega \ldots \cdot \mathrm{d}\boldsymbol{\omega}$, is an integral over a hemisphere of inward directions.

$L(\boldsymbol{\omega})$, is light coming inward toward $\boldsymbol{p}$ from direction $\boldsymbol{\omega}$.

$\boldsymbol{n}$, is normal direction at $\boldsymbol{p}$.

In order to render geometries with the illuminated results of this material, the following settings are adopted. We assume the shape with this material is denoted by s and its geometric type is of CIndexTriangleSet. The texture mapping on the surface is denoted by $\boldsymbol{T}$: $R^3 \rightarrow R^2 = \boldsymbol{U}$. $\boldsymbol{U}$ stands for the texture space.

*(s.m_pAppearance->m_pTexture->m_Textures[0]) = M_{diffuse}, which stores the diffuse coefficient texture;

*(s.m_pAppearance->m_pTexture->m_Textures[1]) = $M_{\text{irradiance}}$, which stores the irradiance on the surface, i.e. the texel at $\boldsymbol{u}$ $M_{\text{irradiance}}(\boldsymbol{u}) = \int_\Omega L(\boldsymbol{\omega}) \cdot (\boldsymbol{p}) \cdot (-\boldsymbol{\omega} \cdot \boldsymbol{n}) \mathrm{d}\boldsymbol{\omega}$, where $\boldsymbol{u} = \boldsymbol{T}(\boldsymbol{p})$.

s.m_pAppearance->m_pTexture->m_Mode[0] = MODE_REPLACE;
s.m_pAppearance->m_pTexture->m_Mode[1] = MODE_MODULATE;
s.m_pAppearance->m_pTexture->m_Source[0] = SOURCE_DEFAULT;
s.m_pAppearance->m_pTexture->m_Source[1] = SOURCE_DEFAULT;

*(s.m_pGeometry->m_pTexCoord) = $\{\boldsymbol{u}_k \mid \boldsymbol{u}_k$ is the texture coordinate at vertex $\boldsymbol{p}_k$, which satisfies $M_{\text{diffuse}}(\boldsymbol{u}_k) = k_{\text{diffuse}}(\boldsymbol{p}_k)$, $M_{\text{irradiance}}(\boldsymbol{u}_k) = \int_\Omega L(\boldsymbol{\omega}) \cdot (\boldsymbol{p}_k) \cdot (-\boldsymbol{\omega} \cdot \boldsymbol{n}) \mathrm{d}\boldsymbol{\omega}\}$

It actually computes $I(\boldsymbol{p}) = M_{\text{diffuse}}(\boldsymbol{T}(\boldsymbol{p})) * M_{\text{irradiance}}(\boldsymbol{T}(\boldsymbol{p}))$. We need to pre-compute $M_{\text{irradiance}}$.

(3) Inter-reflection and specular reflection considered with diffuse texture.

The rendering equation of the point $\boldsymbol{p}$ on the surface with this material is:

$$I(\boldsymbol{p},\boldsymbol{v}) = k_{\text{diffuse}}(\boldsymbol{p}) \cdot \int_{\Omega} L(\boldsymbol{\omega}) \cdot (\boldsymbol{p}) \cdot (-\boldsymbol{\omega} \cdot \boldsymbol{n}) \, \mathrm{d}\boldsymbol{\omega} + k_{\text{reflection}} \cdot L(\boldsymbol{p},\boldsymbol{v}) \qquad (3.4)$$

where $\boldsymbol{v}$ is the viewpoint. $L(\boldsymbol{p},\boldsymbol{v})$ is the ray from $\boldsymbol{v}'$ to $\boldsymbol{p}$ direction, and $\boldsymbol{v}'$ is the viewpoint's symmetric point about $\boldsymbol{p}$.

The other symbols can be referred to the above rendering equation where inter-reflection is considered with diffuse material.

In order to render geometries with the illuminated result of this material, the following settings are adopted. We assume the shape of this material is denoted by s and its geometric type is of CIndexTriangleSet. The texture mapping on the surface is denoted by $\boldsymbol{T}$: $R^3 \rightarrow R^2 = \boldsymbol{U}$. $\boldsymbol{U}$ stands for texture space.

*(s.m_pAppearance->m_pTexture->m_Textures[0]) = M_{diffuse}, which stores the diffuse coefficient texture

*(s.m_pAppearance->m_pTexture->m_Textures[1])=$M_{\text{irradiance}}$, which stores the irradiance on the surface, i.e.

the texel at $\boldsymbol{u}$ $M_{\text{irradiance}}(\boldsymbol{u}) = \int_{\Omega} L(\boldsymbol{\omega}) \cdot (\boldsymbol{p}) \cdot (-\boldsymbol{\omega} \cdot \boldsymbol{n}) \, \mathrm{d}\boldsymbol{\omega}$, where $\boldsymbol{u} = \boldsymbol{T}(\boldsymbol{p})$.

*(s.m_pAppearance->m_pTexture->m_Textures[2])=$M_{\text{reflection}}$, which is the CComposedCubeMapTexture instance.

s.m_pAppearance->m_pMaterial->m_Reflectance = $k_{\text{reflection}}$;
s.m_pAppearance->m_pTexture->m_Usage[0] = USAGE_DIFFUSE_MAP;
s.m_pAppearance->m_pTexture->m_Usage[1] = USAGE_LIGHT_MAP;
s.m_pAppearance->m_pTexture->m_Usage[2] = USAGE_REFLECTION_MAP;
*(s.m_pGeometry->m_pTexCoord) = {$\boldsymbol{u}_k$ | $\boldsymbol{u}_k$ is the texture coordinate of $\boldsymbol{p}_k$, which satisfies $M_{\text{diffuse}}(\boldsymbol{u}_k) = k_{\text{diffuse}}(\boldsymbol{p}_k)$, $M_{\text{irradiance}}(\boldsymbol{u}_k) = \int_{\Omega} L(\boldsymbol{\omega}) \cdot (\boldsymbol{p}_k) \cdot (-\boldsymbol{\omega} \cdot \boldsymbol{n}) \, \mathrm{d}\boldsymbol{\omega}$ }
s.m_pShader-> m_Language = LANGUAGE_CG;
s.m_pShader->m_bActiveFlag = false;
s.m_pShader->m_ShaderPrograms[0]-> m_ProgramType = PRGTYPE_VERTEX;
s.m_pShader->m_ShaderPrograms[0]->m_SystemVariables[] =
{ "System_ModelMatrix", "System_ModelViewProj", "System_EyePosition" };
s.m_pShader->m_ShaderPrograms[0]->m_URL =
{ "GenerateCubeMapReflectionDirection.cg" };
s.m_pShader->m_ShaderPrograms[1]-> m_ProgramType =
PRGTYPE_FRAGMENT;
s.m_pShader->m_ShaderPrograms[1]->m_SystemVariables[] =
{ "Shape_DiffuseMap", "Shape_LightMap", "Shape_ReflectionMap",
"Shape_Reflectance" };
s.m_pShader->m_ShaderPrograms[1]-> m_URL =
{ "Diffuse_Interreflection_Mirror.cg" };

The vertex shader program GenerateCubeMapReflectionDirection.cg produces the reflection directions in environment mapping. The fragment shader program Diffuse_Interreflection_Mirror.cg blends three textures, i.e.,

$M_{\text{diffuse}} * M_{\text{irradiance}} + k_{\text{reflectance}} * M_{\text{reflection}}$.

Since these shader program codes are quite common, we do not list them here.

3.6 Scene Graph

A scene graph is represented by a tree structure. The classes for building the tree structure are shown in Fig. 3.20.

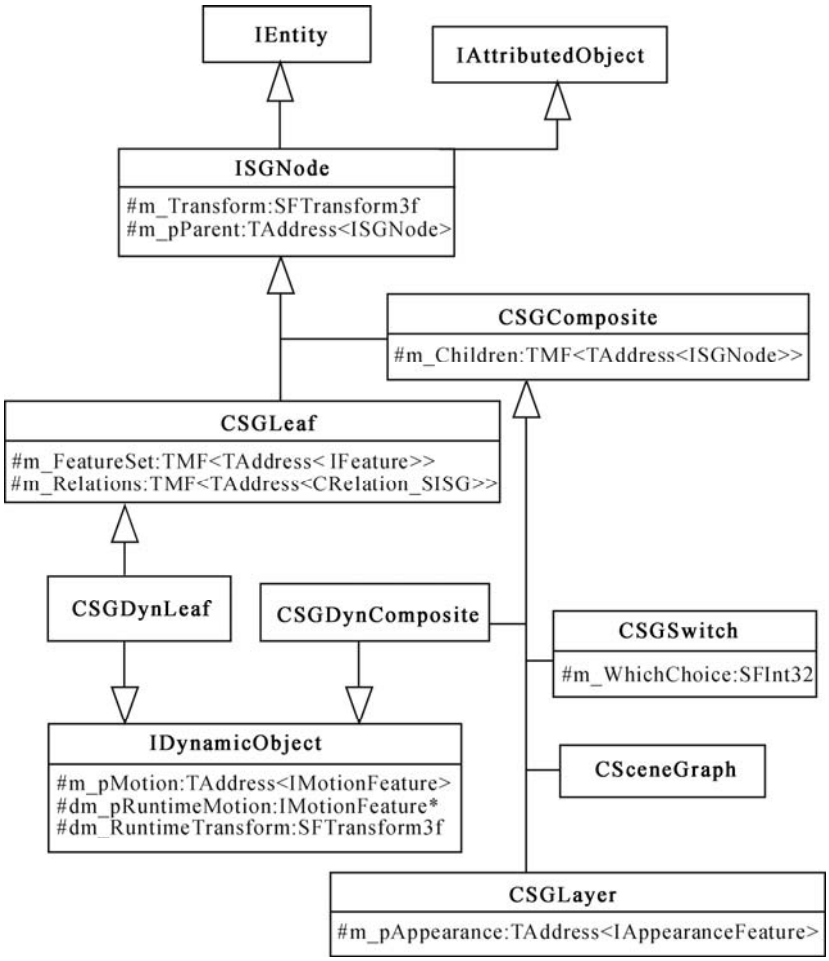

Fig. 3.20 The classes related to scene graph

.

Here, we still adopt the composition design pattern to build the scene graph. Since the ISGNode inherits IEntity, it has a persistent characteristic. The data members of the ISGNode include: one 3D transformation m_Transform and one TAddress<ISGNode> m_pParent pointing to its parent node. Given an ISGNode instance (either CSGLeaf or CSGComposite instance), by using m_Transform the coordinate frame of the child nodes (as CSGComposite::m_Children) or the contained features (as CSGLeaf::m_FeatureSet) can be transformed to the ISGNode instance's coordinate frame. The transform hierarchy can be achieved by

this means. Each node in the Visionix scene graph is unique, i.e., any ISGNode instance cannot be referenced in any scene graph.

CSGComposite represents the middle node of a scene graph. The data member m_Children is an array of TAddress<ISGNode>, and each element stands for a child node.

CSceneGraph represents the root node of any scene graph. Therefore we exploit it to represent a scene graph.

CSGLayer represents the kind of middle nodes that have certain appearance properties. It has the data member m_pAppearance pointing to an IAppearanceFeature instance, indicating all descendant nodes inheriting the same appearance specified by the IAppearanceFeature instance.

CSGSwitch represents the kind of middle nodes whose children nodes are feasible models for the CSGSwitch node. Hence, at most one child node in CSGSwitch::m_Children represents the valid model at any time, and the index of the valid child node is specified by CSGSwitch:: m_WhichChoice.

CSGLeaf is the type of leaf node. In a scene graph, one leaf node represents one object. Every leaf node, as a CSGLeaf instance, contains an array of TAddress<IFeature>, named by m_FeatureSet. Each element in m_FeatureSet points to a feature belonging to the leaf node. These features are regarded as the leaf node's features.

From the viewpoint of class definition, there is no specification for which kind of features can be contained in CSGLeaf::m_FeatureSet, nor for the relations between the features in the set. This gives the scene graph great flexibility.

On the other hand, due to this flexibility, we do not derive ISGNode from IBoundedObject. The reason is that if an ISGNode instance contains an unbounded feature, like a point light, it would be meaningless to say that the ISGNode instance is an IBoundedObject.

In practice, we often come across a situation where objects in a scene graph should be represented in different forms for different operations. Traditional scene graphs are unable to meet this demand. In Visionix, we can exploit CSGLeaf::m_FeatureSet to make one object have multiple representations. In this case, each feature in a leaf node's m_FeatureSet is just one kind of representation for the same object corresponding to the leaf node. These features describe one object in different ways. For example, for a building, 2D plot, 3D structure model, 3D facade model, or photo image could be the building's representation. These representations can be stored in the feature set of one CSGLeaf instance. Of course, m_FeatureSet could be used in other ways. It depends on the schema of the scene model, which is introduced in Section 3.8 "Scene Model Schema".

In CSGLeaf, there is an important data member m_Relations, an array of TAddress<CRelation_SISG>. CRelation_SISG is used to depict the relation between the scene graph leaf nodes and spatial indices. We are going to introduce CRelation_SISG in detail in the following section.

CSGDynLeaf represents left nodes with some dynamic behaviors. It inherits from both IDynamicObject and CSGLeaf. In IDynamicObject, the data member m_pMotion points to an IMotionFeature instance, which is for describing the possible motion of the corresponding leaf node. IMotionFeature is the base class

for describing any motions. The data member dm_pRuntimeMotion also points to an IMotionFeature instance, which is for describing the motion of the leaf node at runtime. Since m_pMotion points to the initial motion, the pointer should be persistent. Therefore, the type of m_pMotion is TAddress<IMotionFeature>. The type of dm_pRuntimeMotion is IMotionFeature*, because it is set at runtime and does not need to be recorded when application exits. If a dynamic leaf node only has one motion in its whole lifetime, m_pMotion and dm_pRuntimeMotion must point to the same IMotionFeature instance. Nevertheless, a dynamic leaf node has several motions. The instances pointed by m_pMotion and dm_pRuntimeMotion may not be the same.

3.7 Spatial Index

A spatial index is also represented by a tree structure. Each node in a tree-like spatial index represents a space block. The node hierarchy represents the hierarchy of the corresponding space blocks. Any middle node represents a space block composed of the smaller space blocks represented by its children nodes. The classes representing the tree structure are shown in Fig. 3.21.

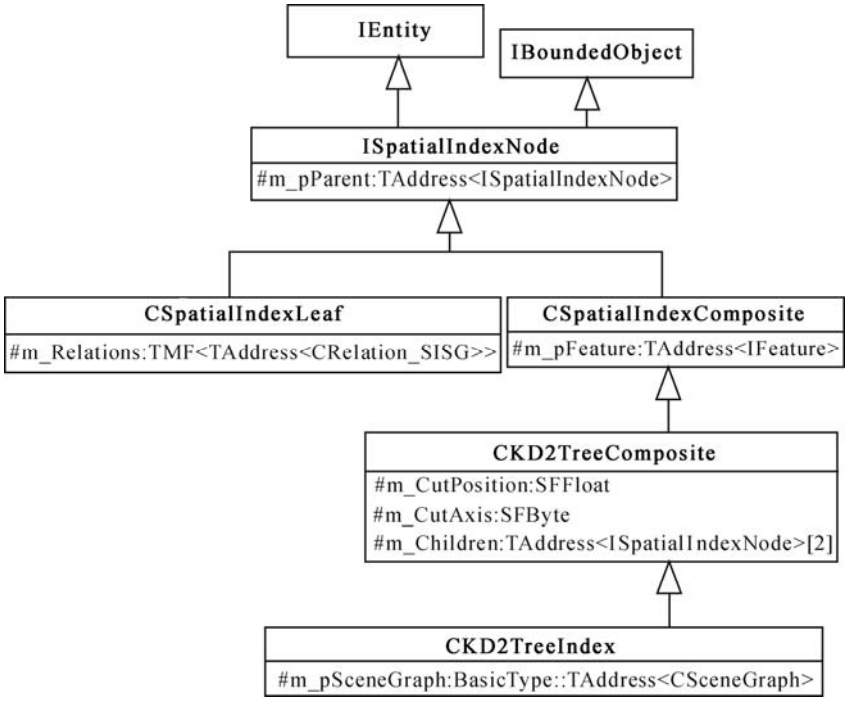

Fig. 3.21 The classes related to spatial index

ISpatialIndexNode is the base class for nodes of a tree-like spatial index, and inherits IEntity and IBoundedObject. It means that the nodes are persistent and of bounded volume. However, the meaning of the bounding box specified by the data members in IBoundedObject is dependent on the scene model schema, which is stated in Section 3.8 “Scene Model Schema”. The bounding box can be the space block of this node, or be the bounding box of the features contained in the space block of the node.

CSpatialIndexLeaf is for the leaf node and CSpatialIndexComposite is for the middle node. CKD2TreeComposite is the subclass of CSpatialIndexComposite and represents the middle node of the KD-tree, which is a popular kind of spatial index in computer graphics. By CSpatialIndexLeaf and CKD2TreeComposite, a KD-tree can be represented. Other spatial indices can be represented similarly.

Like scene graph, we adopt a composition design pattern to represent the spatial index. CKD2TreeComposite::m_Children has two TAddress<ISpatialIndexNode> pointers pointing to left child node and right child node. The base class ISpatialIndexNode also provides a pointer pointing to the node’s parent node.

CKD2TreeComposite::m_CutAxis depicts the axis along which the node’s space block is cut into two parts. In other words, the normal of the cutting plane is aligned with the axis. m_CutPosition records the position on which the cutting plane passes through the axis.

CSpatialIndexComposite::m_pFeature points to a feature, the meaning of which should be specified by the scene model schema, and which usually represents a composition of the objects indexed by the descendant nodes in this node. If someone wants to represent hierarchical levels of detail based on this KD-tree structure, m_pFeature could be used to point to a CStaticLOD feature, which represents the levels of detail of the objects indexed by this node’s descendant.

In CSpatialIndexLeaf, m_Relations is an array of TAddress<CRelation_SISG>. CRelation_SISG represents the relation between feature and space. Fig. 3.18 is a class diagram showing the relations among CRelation_SISG, CSGLeaf, CSpatialIndexLeaf and IFeature. In short, we call CRelation_SISG instance relation. In the diagram, any association with multiplicity=1 is mapped to TAddress<…> in C++, any association with multiplicity=1..* or 0..* is mapped to TMF<TAddress<…>>. From the class diagram we obtain the following facts:

(1) One CSpatialIndexLeaf node refers to one or more relations, and each relation refers to one CSpatialIndexLeaf node.

(2) One CSGLeaf node refers to zero or more relations, and each relation refers to one CSGLeaf node and one feature. The feature belongs to the CSGLeaf node.

CRelation_SISG is a bridge connecting CSGLeaf and CSpatialIndexLeaf. In fact, through CRelation_SISG, we can define several relationships between the scene graph and spatial index (Fig. 3.22). Each relationship is called relation schema, which is discussed in the following subsection.

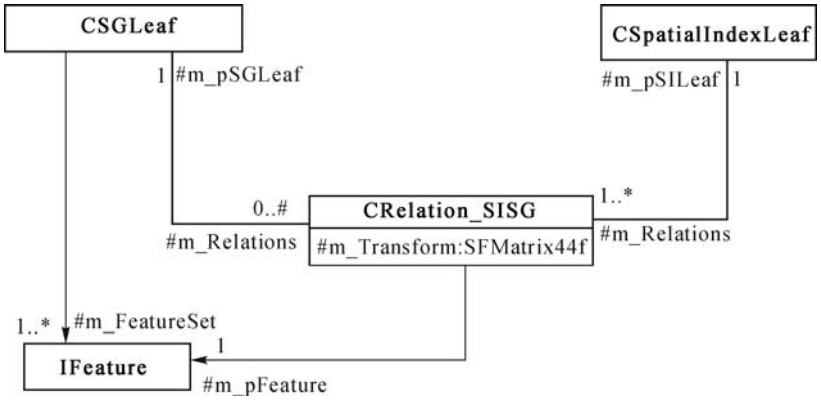

Fig. 3.22 The classes related to the spatial index

The data member CRelation_SISG::m_Transform is a 4×4 matrix, which transforms the local coordinate frame of the feature pointed by m_pFeature to the world coordinate frame.

3.7.1 Relation Schema A

Given a CSpatialIndexLeaf node X, it refers to one or more relations $\{R_i \mid i = 1,2,\ldots,N\}$. For relation R_i, R_i.m_pFeature points to a feature F_i and R_i.m_pSGLeaf points to the corresponding leaf node G_i in the scene graph. It means that there are several features $\{F_i \mid i=1,2,\ldots,N\}$ in the space block of the spatial index node X. In other words, a single spatial index leaf node corresponds to multiple scene graph leaf nodes. By this means, given any spatial index leaf node, the corresponding scene graph leaf nodes or the corresponding features can be retrieved in time $O(1)$.

Given a CSGLeaf node G, for convenience we assume there is only one feature F in G.m_FeatureSet. G.m_Relations contains several relations $\{R_i \mid i=1,2,\ldots,M\}$. For relation R_i, R_i.m_pFeature points to the feature F in G.m_FeatureSet and R_i.m_pSILeaf points to the spatial index leaf node, whose space block contains some part of F. If there is more than one feature in G.m_FeatureSet, each CRelation_SISG instance links one feature and its corresponding spatial index leaf node. Thus, given a scene graph leaf node, the corresponding space blocks can be retrieved in time $O(1)$.

The object diagram in Fig. 3.23 provides an example. There are two scene graph leaf nodes, G1 and G2. Each node has two features and two relations. The spatial index leaf node X1 refers to G1.m_Relations[0], which refers to the feature G1.m_FeatureSet[0] and the scene graph node G1. Another spatial index leaf node X2 refers to three relations, G1.m_Relations[1], G2.m_Relations[0] and G2.m_Relations[1]. G1.m_Relations[1] refers to G1.m_FeatureSet[1] and G1,

G2.m_Relations[0] refers to G2.m_FeatureSet[0] and G2, and G2.m_Relations[1] refers to G2.m_FeatureSet[1] and G2.

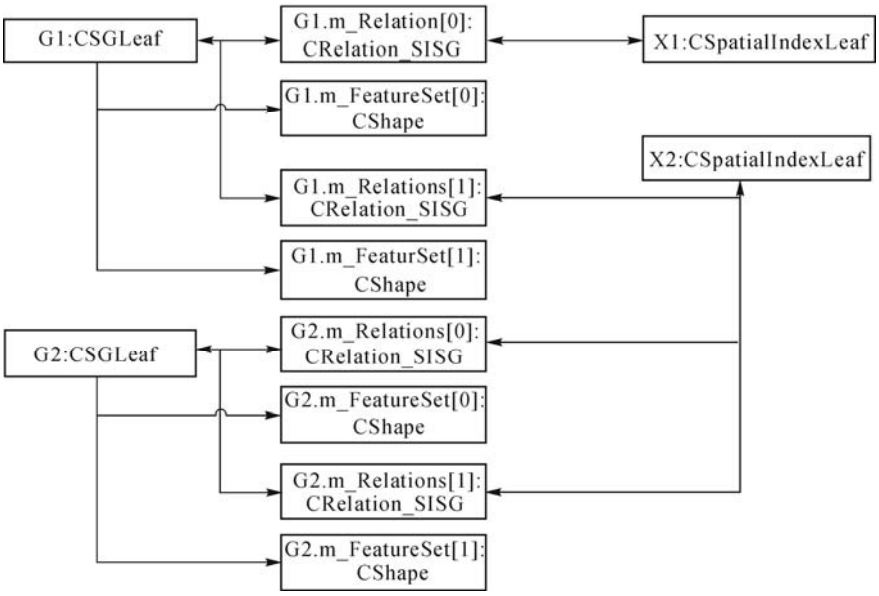

Fig. 3.23 The object diagram for an example of relation schema A

The advantage of the Relation Schema A is that it is intuitive and simple. It is consistent with the common understanding of the relationship between scene graph and spatial index. Nevertheless, there is one limitation, as each relation must refer to the feature that is contained in m_FeatureSet of one scene graph leaf node. When the feature is a CSuperShape or CComposeShape instance, the corresponding relation has to refer to the CSuperShape or CComposeShape instance, and is unable to refer to its sub-shape or child shape. This means the spatial index cannot index any sub-shapes or child shapes. It leads to the coarser granularity of indexed objects, and further reduces effectiveness of the spatial index.

For example, a scene graph leaf node G contains a CSuperShape instance F, which consists of 50 sub-shapes. F has huge volume, so it appears in the space blocks of 50 spatial index leaf nodes. In rendering, the spatial index is adopted to accelerate view frustum culling. If one of the 50 spatial index leaf nodes appears in the view frustum, F is believed to be potentially visible and all sub-shapes of F have to be rendered. However, there is in fact only one sub-shape that appears in the view frustum. Thus, if the links between sub-shapes and spatial index leaf nodes could be built, visibility culling would be performed for the sub-shape instead of the entire CSuperShape instance. That is the problem to be solved by Relation Schema B.

3.7.2 Relation Schema B

In Relation Schema B, the feature referred to by a relation need not be the feature in m_FeatureSet of a scene graph left node. In most cases of Relation Schema B, the relation refers to one sub-feature of the feature in m_FeatureSet.

In Fig. 3.24 an example is shown. The scene graph leaf node G1 has one CSuperShape instance, G1.m_FeatureSet[0] and two relations. Each relation refers to one of the sub-shapes of the CSuperShape instance and a spatial index leaf node.

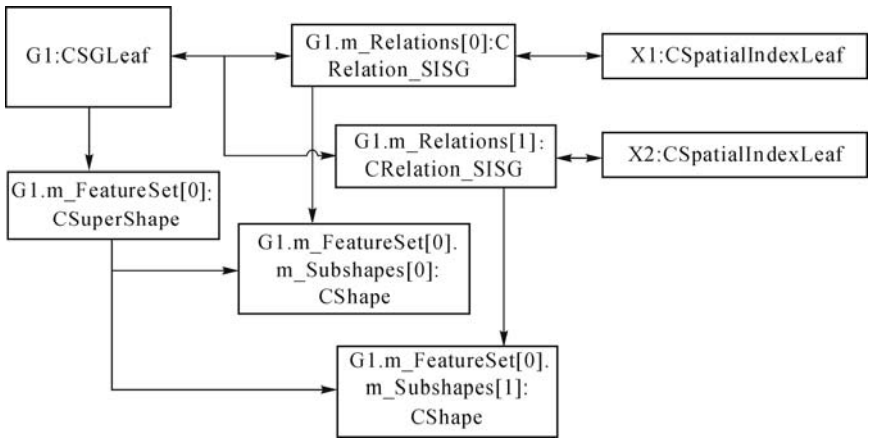

Fig. 3.24 The object diagram for an example of relation schema B

If more than one feature in m_FeatureSet has its own spatial indices, the relations in m_Relations refer to the leaf nodes of different spatial indices. In Fig. 3.25 an example is shown. In this example, there are two KD trees, Ta and Tb. There is no problem to find corresponding sub- features from any spatial index leaf node Xa1, Xa2, Xb1 or Xb2. Nevertheless, it is difficult to find the spatial index leaf nodes corresponding to G1.m_FeatureSet[0] or G1.m_FeatureSet[1], since m_Relations in the scene graph leaf node have no relation to G1.m_FeatureSet[0] or G1.m_FeatureSet[1]. To handle this problem, we set up a lookup table to map the features in m_FeatureSet and the relations in m_Relations. This lookup table can be an attribute of the scene graph leaf node. Remember CSGLeaf is also a subclass of IAttributedObject.

3.8 Scene Model Schema

So far, we have introduced a set of data structures, and by utilizing them we can construct a variety of scene models. Obviously, it is just the design purpose for these data structures.

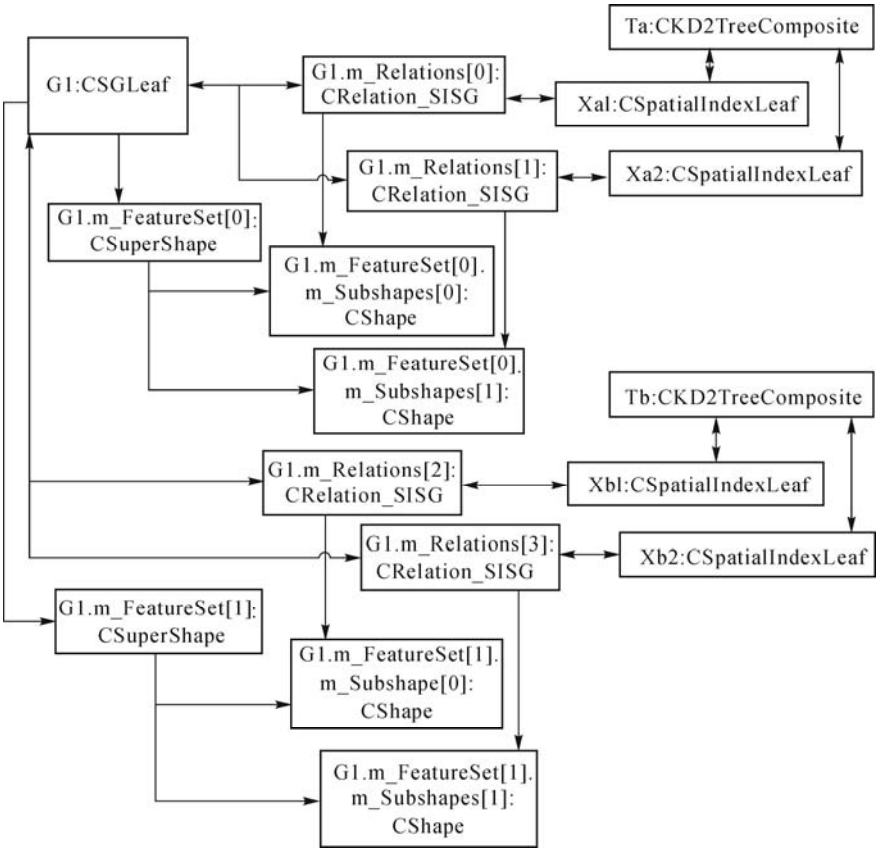

Fig. 3.25 The object diagram for another example of relation schema B

However, from the viewpoint of developers who develop applications based on scene models, it is better to have a schema that is a specification which states the organization of a scene model, and which is the guidance for developers to write appropriate codes to find the desired data in a scene model. The schema applies a certain constraint so as to reduce the uncertainties in scene model representation, which further reduces the difficulties in application developments and improves application runtime efficiency.

For example, if we knew each CSGLeaf node has three features, and knew the first one is for a 2D contour model, the second one is for a 3D mesh model and the third one is a photo image representation, it would be much easier for us to write codes handling CSGLeaf nodes, so as to save a lot of checks. If a 2D contour is required to be drawn, only the first features in all CSGLeaf nodes should be accessed. If a 3D model is required to be drawn, only the second features should be accessed. If we had no idea of this, it would be rather difficult to write codes handling CSGLeaf nodes. Developers should classify the features according to their types, names, data members or other criteria, and this classification could not

be done for all purposes.

Based on this consideration, we introduce the concept of a scene model schema in the Visionix system. It gives a set of constraints and classifications for objects in the scene model. When a scene model passes the examination of a certain scene model schema, developers can write codes based on this scene model schema.

In the remaining part of this section, we show an example, which is defined for an urban planning application. Here, we state the scene model schema in natural language instead of formal representation.

(1) The root of the scene graph has only two nodes. Node [_Rendering_] is the root node of the subtree for rendering, while Node [_NoRendering_] is the root node not for rendering.

(2) Node [_Rendering_] has seven CSGLayer nodes, which are

- Node [_Layer_Terrain_], representing terrains,
- Node [_Layer_Building_], representing buildings,
- Node [_Layer_Water_], representing water systems,
- Node [_Layer_Vegetation_], representing plants,
- Node [_Layer_Road_], representing roads,
- Node [_Layer_Background_], representing backgrounds,
- Node [_Layer_Dynamic_]， representing dynamic objects.

(3) Node [_NoRendering_] has one CSGLayer node, Node [_Layer_ScenicSpot_], representing user predefined viewpoints.

(4) The CSGLayer nodes appear and only appear in node [_Rendering_] and node [_NoRendering_].

(5) The transform in CSGLayer node is identity. Each CSGLayer has zero or many CSGComposition nodes. Each CSGComposition node has zero or many CSGComposition/CSGLeaf nodes. Of course, any leaf node under Node [_Layer_Dynamic_] is CSGAnimLeaf node.

(6) From Node [_Rendering_] to bottom, each CSGLeaf node contains two features. The first feature is a 3D model and the second one is a 2D model. Both represent the same object in space.

- The 3D model is represented by CComposedShape or CStaticLOD. CComposedShape cannot be nested. Each level of detail in CStaticLOD is represented by CSuperShape.
- The 2D model is represented by CSuperShape.
- CMultiTextureCoordinate cannot be nested.

(7) Spatial index follows the following regulations.

- Using Relation Scheme B;
- The features referred to by CRelation_SISG instance are CShape, CBillboard, or CStaticLOD instance.
- The bounding box of the spatial index node is the bounding box of all features indexed by the node through relations.
- There are CRelation_SISG instances in CDynSGLeaf and CDynSGComposite nodes. In other words, do not index dynamic scene graph nodes.

3.9 Scene Model Interface and Implementation

In the Visionix system, the scene model is considered a container of various entities. Through the previous introduction, it is known that there are four categories of entities: feature (IFeature subclasses), scene graph node (ISGNode subclasses), spatial index node (ISpatialIndex subclasses) and relation (CRelation_SISG).

We have adopted the strategy of separating interface and implementation in scene model design to improve extensibility. We have defined the interface ISceneModel and provided an implementation CSceneModelImp1 for ISceneModel.

3.9.1 Scope of Name and ID

In a Visionix scene model, each entity must own a unique identifier (ID). In a Visionix scene model, the ID is used to locate a corresponding entity either in the host memory or in the external memory (such as disk file). In the Visionix system, there is a lookup table to map every valid ID to the address of the ID's corresponding entity. The Visionix scene model adopts TAddress to realize inter-reference of entities, and one TAddress<entity class> instance contains both ID and address (or say pointer) of an entity. The correspondency between ID and address in a TAddress<entity class> instance is constructed based on the lookup table.

Besides ID, every entity can have a name, which is represented by a string. The entity name must be unique in the entire scene model. It is more user-friendly for people to look up an entity with entity name both in development or in application.

3.9.2 Transaction

The Visionix system supports transaction processing to guarantee integrity of the scene model. So far, only a non-nested transaction is supported. Some functions defined in ISceneModel must be invoked during one transaction.

3.9.3 Scene Storage

The Visionix scene model has all entities stored in an object called *scene storage*. The scene storage provides one virtual storage space, which removes the difference between host memory and external memory.

With scene storage, developers only need to create and process a scene model

in the host memory, and the consistency of the scene model between host memory and external memory will be automatically maintained by the scene storage. In other words, the scene storage encapsulates all operations for the external memory.

The scene storage consists of a set of object tables. Each object table takes charge of storing and managing all instances of one entity class. The concept meaning of object tables is the same as the table in a database system. The object table has the following characteristics:

(1) An object table consists of a set of records. Each record stores an entity.

(2) Creating a new entity in the scene model is equivalent to adding a new record to its corresponding object table. Releasing an entity from the scene model is equivalent to deleting its corresponding record from the object table.

(3) Both object table or record can be loaded into or unloaded from the host memory.

(4) It is zero-copy to convert a record to its stored entity in the host memory.

Object tables are completely transparent for developers. It means that developers can write codes to traverse a scene graph or spatial index in their normal way, without any knowledge of the object table.

Another concept in the Visionix scene model is the storage device. It provides an abstract for various storage media. The typical storage media is the file system. IStorageDevice defines the interface.

From the viewpoint of software architecture, a scene model is built upon scene storage and scene storage is built on a storage device.

3.9.4 Reference and Garbage Collection

In a Visionix scene model, inter-reference between entities via TAddress are very common. It should be emphasized that all entities only refer to each other, and there is no aggregation between them. For instance, in a scene graph or spatial index, any association between node and node or node and feature is a reference. In other words, the life period of each entity is independent of others. For example, when a CGroup instance is deleted, no child feature in the CGroup instance will be deleted automatically.

The advantage of adopting a reference is to reuse the entity as much as possible and to decrease resource consumption. Moreover, since any association between entities is a reference, it will be easy for developers to remember this fact. They do not attempt to distinguish which association is a reference and which is aggregation.

The drawback of this approach is that when an entity is to be deleted all related entities must be carefully checked. Otherwise, the scene model integrity would be damaged with many invalid pointers and useless entities left. The following pseudo code shows such kinds of checks.

```
DeleteEntity(anEntity)
{
  if anEntity is referenced by other entities then
      can not delete anEntity and just return
  else if anEntity does not reference any other entity
      remove anEntity from the corresponding object table and destroy it.
  else
      for each entity e referenced by anEntity
          if e is not referenced by other entities
              DeleteEntity(e)
      remove A from the corresponding object table and destroy it.
}
```

DeleteEntity is a recursive procedure, which is rather cumbersome. To solve the entity deletion problem in a more efficient way, we adopt the garbage collection approach. For a Visionix scene model, if an entity cannot be visited from any scene graph or spatial index through any possible inter-reference, it is regarded as garbage. As long as the function ISceneModel::DeleteUnrefEntities() is invoked, all garbages in the scene model are automatically collected and released. By using garbage collection, there is no need to explicitly delete any entity, which involves rather complicated inter-reference circumstances. The only thing that developers should do is to invoke DeleteUnrefEntities at the proper time to start the garbage collection.

3.9.5 Data Visit and Cache

A Visionix scene model exploits many similar techniques, which are widely used in an ordinary database, to realize rapid access and updating of a large-scale dataset. When the size of the scene model becomes tens or hundreds of giga bytes, loading all data into the host memory seems impractical. Thus, a Visionix scene model adopts out-of-core data management to provide external memory data seeking, buffering and updating.

There are two modes for visiting an entity in a Visionix scene model:

(1) Access to entities while traversing a scene graph or spatial index from a certain node in a certain order.

(2) Random access to entities by their IDs or names.

To realize the visiting mode (a), a Visionix scene model keeps some parts of scene graphs and spatial indices in a cache. Moreover, during the traverse, the system predicts which nodes and related entities are going to be visited, and prefetches them in the cache. The Least Recent Unused (LRU) strategy is adopted to discard unused entities from the cache.

To realize the visiting mode (b), a Visionix scene model maintains another cache in the scene storage. To respond to a random accessing request, the system

first looks up the entity in the cache. If the corresponding entity does not exist there, the system searches the external memory address of the entity through a lookup table and then seeks the entity according to the address in the external memory. Afterwards, the entity is loaded and placed in the cache. The same LRU strategy is adopted to manage the cache.

Loading the entity in the above two ways is transparent for developers.

However, in practice, there are some entities that contain large data themselves. Moreover, in many cases, only a small part of an entity is required to be loaded in the host memory. Thus, a Visionix scene model provides a way to load or unload the parts of some entities. These entities are called out-of-core entity, which will be introduced in the following subsection 3.9.6 “Out-of-Core Entity”. In an out-of-core entity, there are two kinds of data. One is called the in-core part, the other is called the out-of-core part. When an entity is required to be visited, only the in-core part of the entity will be loaded automatically, as stated in the above two modes. Loading the out-of-core part is postponed until the out-of-core part is verified to be really required. The function ISceneStorage::LoadEntityOutOfCorePart and ISceneStorage::UnloadEntityOutOfCorePart are used to do the loading and unloading respectively.

3.9.6 Out-of-Core Entity

Fig. 3.26 shows a class diagram of the out-of-core entity classes. All out-of-core entity classes inherit both IOutOfCoreObject and corresponding entity base classes. Meanwhile, IOutOfCoreObject is the subclass of IThreadSafeObject. IThreadSafeObject contains a mutex for access control in a multi-threading computing enviroment. For an out-of-core entity, the mutex can be used to control accessing, loading or unloading the out-of-core part by multiple threads.

IOutOfCoreObject provides function IsInHostMemory to query whether the out-of-core part is in the host memory.

A Visionix scene model provides an out-of-core version of CColor, CColorRGBA, CCoordinate, CIndex, CNormal and CTextureCoordinate, CTextureCoordinate3D and CImage class. The naming rule for the out-of-core version is to add “OutOfCore” to their class names. The reason why we provide an out-of-core version for these entities is that the total data size of these entities is usually much larger than others for an ordinary scene. Controling the memory occupation of these entities is more useful for achieving better balance between development complexity and runtime effectiveness.

If we consider an image as an array of pixels, the type of all out-of-core entities is in fact an array. We decompose an array into two parts. The first one is control information, including element number, element type, etc. The second one is a buffer containing elements. Obviously, the first part is the in-core part, and the second one is the out-of-core part. In most cases, the size of the buffer is much larger than the control information.

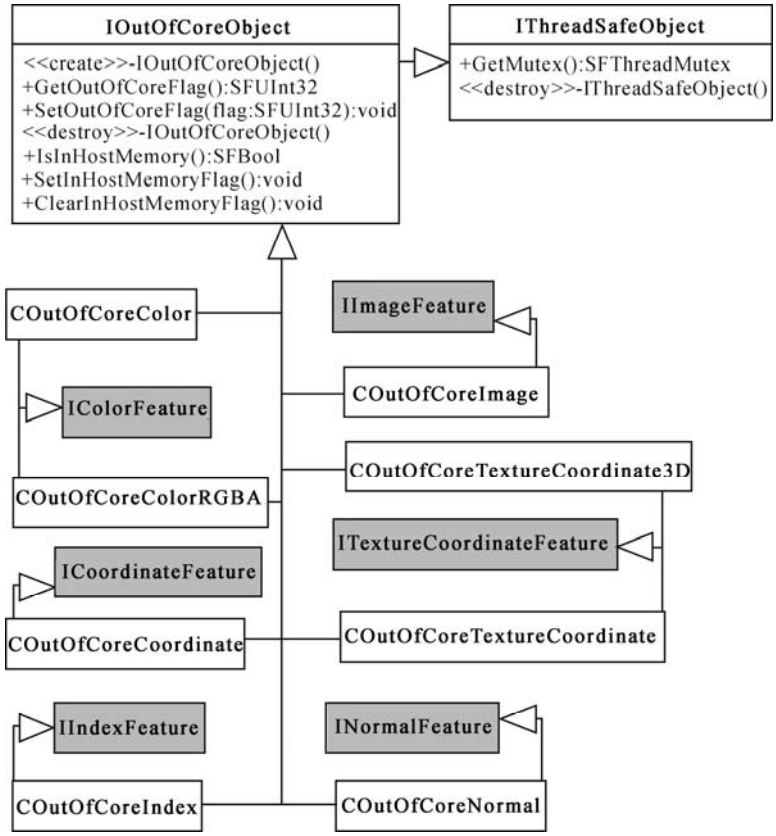

Fig. 3.26 The classes for out-of-core entities

3.9.7 ISceneModel

ISceneModel is an abstract class, without any data member. Intuitively, any object with ISceneModel interface can be regarded as a composition of one scene storage, several scene graphs and several spatial indices.

The functions of ISceneModel can be classified into several categories as follows.

3.9.7.1 Scene Model Related Functions

```
SFByte OpenSceneModel(const SFString& SceneModelName, const SFInt mode)
```

Open scene model with name specified by the input parameter SceneModelName.

The input parameter “mode” specifies the cache mode after opening the scene model. The value of the mode can be one of the following enumeration values.

(1) CACHE_COMPLETE_OBJECT: specifying the system should put both in-core and out-of-core parts in the cache.

(2) CACHE_OBJECT_INCORE_PART: specifying the system should only put the in-core part in the cache.

The default cache mode is CACHE_OBJECT_INCORE_PART. For a non out-of-core entity, the entire entity is put in the cache.

```
SFBool CloseSceneModel()
```

Close the scene model.

```
SFBool CreateSceneModel(const SFString& SceneModelName,
                        const SFString& FirstSceneGraphName,
                        const SFString& FirstSpatialIndexName)
```

Create a new scene model with a blank scene graph and a blank spatial index. Function callers should provide three names for the scene model, first scene graph and first spatial index respectively. The names should not be empty. If multiple scene graphs (or spatial indices) are required, developers should use the member function CreateGraph (CreateIndex) to add other scene graphs (or spatial indices).

The scene model is created on a storage device which shares the name with the scene model. If the storage device has not been created yet, this function creates it in default fashion.

```
SFBool IsOpened()
```

Query whether the scene model is opened.

3.9.7.2 Scene Graph Related Functions

```
SFInt CreateGraph(const SFString& SceneGraphName)
```

This function should be used in a certain transaction.

It creates a blank scene graph with the name specified by the input parameter SceneGraphName. The name must be non-empty, and must be unique in the whole name space of the scene model.

If the transaction is rolled back, any outcomes of this function will be canceled.

```
SFUInt GetGraphCount() const
```

Query the number of scene graphs in the scene model.

```
CSceneGraph* GetGraph(SFInt i)
```

Get the pointer which points to the i-th scene graph. The index of the first scene graph is 0.

```
SFBool ForceDeleteGraph(SFInt index)
```

This function should be used in a certain transaction.

This function forcely deletes the *i*-th scene graph. Since any spatial index object has a reference to a scene graph that owns this spatial index, when the scene graph is forcely deleted, the corresponding spatial index must reset the scene graph reference, or just delete the spatial index.

If the transaction is rolled back, any outcomes of this function will be canceled.

3.9.7.3 Spatial Index Related Functions

```
SFInt CreateIndex(const SFString& IndexName)
```

This function should be used in a certain transaction.

It creates a blank spatial index with the name specified by input parameter IndexName. The name must be non-empty, and must be unique in the whole name space of the scene model.

If the transaction is rolled back, any outcomes of this function will be canceled.

```
SFUInt GetIndexCount() const
```

Query the number of spatial indices in the scene model.

```
CKD2TreeIndex* GetIndex(SFInt i)
```

Get the pointer which points to the *i*-th spatial index. The index of the first spatial index is 0.

```
SFBool ForceDeleteIndex(SFInt index)
```

This function must be used in a certain transaction.

This function forcely deletes the *i*-th spatial index, regardless of the fact that it is referred to by others.

If the transaction is rolled back, any outcomes of this function will be canceled.

3.9.7.4 ISceneStorage Related Function

```
SceneStorage::ISceneStorage* GetSceneStorage()
```

Get the interface of the scene strorage. Many important operations, for example, to begin or end a transaction, to create, delete or save an entity, must be done through the ISceneStorage interface. ISceneStorage will be introduced in subsection 3.9.8 “ISceneStorage”.

3.9.7.5 Other Functions

```
void *Pointer(SFID ID)
```

Change an entity’s ID to a pointer that points to the entity.

```
SFBool DeleteUnrefEntities()
```

MUST NOT be called in any transaction.

Perform garbage collection, release both host memory and external memory occupied by garbages. If an entity cannot be visited by traversing all possible references from the roots of all scene graphs or spatial indices, the entity is garbage.

It can lead to writing the data to external memory, so it is better not to call this function frequently.

Also, see the sub section 3.9.4 “Reference and Garbage Collection”.

```
void SetDeviceDirectory(const SFString& dir)
```

Set the position of the storage device. This function should be called before calling CreateSceneMode.

```
const SFString& GetDeviceDirectory() const
```

Query the position of the storage device.

3.9.8 ISceneStorage

ISceneStorage is the interface of any scene storage object. There are many important operations done by ISceneStorage. The important functions of ISceneStorage are listed as follows.

3.9.8.1 Transaction Related Functions

```
SFInt BeginTransaction()
SFBool CommitTransaction(SFInt ti)
SFBool RollbackTransaction(SFInt ti)
```

The return value of BeginTransaction is the transaction identifier. This identifier is the input parameter passing into CommitTransaction and RollbackTransaction.

The three functions are used to start, commit or roll back a transaction. The time when the transaction starts is the time just after calling BeginTransaction. The calling of both RollbackTransaction and CommitTransaction causes the termination of a transaction. Some functions need to be performed in transaction, and some need not.

Calling RollbackTransaction takes the current state of the scene model back to the state before calling BeginTransaction.

Only when CommitTransaction is called are all updates on the scene model performed in the transaction saved persistently.

3.9.8.2 Entity Related Functions

```
SFID CreateEntity(SFInt32 EntityType,SFString& Name = "")
```

This function must be used in a certain transaction.

This function creates a new entity instance, whose type is specified by the input parameter EntityType, and the name is specified by the input parameter Name.

Note: This function only creates an entity instance, but does not link it to any scene graph of the scene index. It means that after calling CreateEntity, this newly created entity is not referred to by any others. In other words, the entity is garbage and will be released by calling ISceneModel::DeleteUnrefEntities.

A Visionix scene model gives each entity type an enumeration value, all of which are defined in the header file AllEntityTypes.h.

```
SFBool ForceDeleteEntity(SFID ID)
```

This function must be used in a certain transaction.

This function forcely deletes an entity whose identifier is the input parameter ID, regardless of whether the entity is referred to or not. Because a Visionix scene model provides garbage collection, there is no need to call this function frequently.

```
SFBool SaveEntity(SFID ID)
```

This function must be used in a transaction.

This function puts the entity into a queue, where all entities are saved in the external memory when the transaction is committed.

Whenever the content of an entity has been changed, developers should call this function to assure the changes are persistent. If an entity does have changes but developers forget to call SaveEntity to save it, the changes in the entity are uncertain after the transaction is committed.

If many changes take place in an entity, SaveEntity can be called after any

change.

```
SFBool RenameEntity(SFID ID,const SFString& Name)
```

This function must be used in a certain transaction.

Rename the entity whose identifier is the input parameter ID.

The input new name must be unique in the name space of the whole scene model. Otherwise, the function fails and returns a false value.

```
SFString FindEntityName(SFID ID) const
SFID     FindEntity(const SFString& Name) const
```

The two functions are used to find the entity's name and identifier respectively. The name string matching is case sensitive. If the corresponding entity identifier is not found, FindEntity returns –1. However, when FindEntityName returns an empty string, it means either there is no such entity with the specified identifier, or the corresponding entity has no name.

```
SFBool  LoadEntityOutOfCorePart(SFID ID)
SFBool  UnloadEntityOutOfCorePart(BasicType::SFID ID)
```

This function loads and unloads the out-of-core part of the entity. If the entity has no out-of-core part, the function does nothing.

3.9.9 Implementation of ISceneModel and ISceneStorage

So far, we have provided class CSceneModelImp1 and CSceneStorageImp1 as the implementations of ISceneModel and ISceneStorage respectively.

One limitation for the implementations is that they are not thread safe in a general case. Nevertheless, by careful configuration, they can be used in a multi-threading computing environment.

3.9.9.1 CSceneModelImp1

CSceneModelImp1 adopts a 32-bit integer as the data type of the entity identifier. Thus, a scene model represented by CSceneModelImp1 is capable of containing 2G-sized objects.

The data members of CSceneModelImp1 are listed as follows:

```
class CSceneModelImp1:ISceneModel
{
protected:
```

```
    TMF<CSceneGraph*> m_GraphSet;
    TMF<CKD2TreeIndex*> m_IndexSet;
    ISceneStorage* m_pStorage;
    ISceneStorageFactory * m_pStorageFactory;
    SFBool m_bOpenned;
    SFString m_DeviceDirectory;
        …
}
```

m_GraphSet and m_IndexSet are two arrays of pointers, pointing to all CSceneGraph instances and all CKD2TreeIndex instances in the scene model respectively.

m_pStorage points to an ISceneStorage interface.

m_pStorageFactory points to a factory that can create an instance with ISceneStorage interface. Here we adopt the design pattern abstract factory.

m_bOpenned is the flag indicating whether the scene model is opened or not.

m_DeviceDirectory specifies the default storage device directory.

It should be noted that all pointers of these data members are not persistent pointers. It means that CSceneModelImp1 is not persistent. All objects pointed by these pointers are maintained by the ISceneStorage object.

We use the design pattern factory to create instances of ISceneModel. The pseudo codes are listed as follows:

```
class CSceneModelImp1Factory
{
public:
 ISceneModel *Create();
};

ISceneModel * CSceneModelImp1Factory::Create ()
{
    CSceneModelImp1 * pSM = new CSceneModelImp1;
    pSM->m_pStorageFactory = new CSceneStorageImp1Factory;
    return pSM;
}
```

3.9.9.2 CSceneStorageImp1

So far, CSceneStorageImp1 is not thread safe. That is, in a multi-threading computing environment, the member functions of CSceneStorageImp1 should be invoked in only one thread. More precisely speaking, developers must guarantee that, at any time, the member functions of CSceneStorageImp1 are only called in one thread.

However, the applications which use out-of-core entities often require multi-

threading computation. That is, there would be a separate I/O thread that takes charge of loading or unloading the out-of-core parts of entities, and the main thread takes charge of traversing scenes and visiting data. Therefore, it is inevitable for the two threads to access the same out-of-core entity at the same time. Since LoadEntityOutOfCorePart and UnloadEntityOutOfCorePart perform loading and unloading jobs, the I/O thread should lock the target out-of-core entity before calling these two functions, and unlock it after calling. Since each out-of-core entity is also an IThreadSafeObject instance which has a mutex, locking or unlocking an out-of-core enitity can be realized by using the entity's mutex. Meanwhile, the main thread should check the lock state whenever it accesses any out-of-core entity.

The main data members of CSceneStorageImp1 are listed as follows:

```
class CSceneStorageImp1
{
protected:
    TMF<IOTable*>            m_TableSet;
    CAddressTable            m_AddressTable;
    CNameTable               m_NameTable;
    IExMemoryDevice *        m_pDevice;
    …
}
```

m_TableSet is an array of IOTable pointers. Class IOTable is the interface of aforementioned object table. See subsection 3.9.3 "Scene Storage". One object table contains one type of entities. The structure of object tables is similar to that of tables of a relational database. Thus, m_TableSet can be considered as a type-classified container for all entities in the whole scene model.

m_AddressTable is a hashing table, which is used to map entity identifiers to pointers. CSceneStorageImp1 maintains the table on its own.

m_NameTable is also a hashing table, which is used to map entity names to identifiers.

m_pDevice points to a scene storage device, which is represented by the interface class, IExMemoryDevice. CSceneStorageImp1 uses the file system as the concrete scene storage device.

3.10 Scene Manipulator

From the member functions of ISceneModel, it is obvious that all necessary operations to manipulate a scene model are provided through aggregating an ISceneStorage object.

From a practical viewpoint, it is rather inconvenient to use ISceneModel. Hence, the Visionix system provides a scene manipulator object. Here we adopt

the design pattern façade. A scene manipulator is a façade for scene model management, encapsulating a set of scene model operations that are frequently used.

The current version of the Visionix system has provided two kinds of scene manipulators, CSceneModelManipulator_forImportor, CSceneModelManipulator2.

CSceneModelManipulator_forImportor is used for importing external data into Visionix scene models in batch mode. For ordinary applications, CSceneModelManipulator2 is more appropriate. Additionally, developers can develop their own scene manipulators according to their requirements.

CSceneModelManipulator2 is dependent on CSceneModelImp1 and CSceneStorageImp1.

The main functions of CSceneModelManipulator2 are listed as follows.

3.10.1 Manipulator Functions

3.10.1.1 Scene Model Related Functions

```
void SetDeviceDirectory(const SFString& DeviceDirectory)
```

Set the default directory for the scene storage device.

```
ISceneModel *OpenSM(const SFString& name,const SFString& DeviceDirectory="")
```

Open a scene model by its name, and which is on the scene storage with the same name. The input parameter DeviceDirectory specifies the direction of the scene storage. If DeviceDirectory is empty, the default directory is used.

```
SFBool IsOpened() const;
```

Query the open state of the scene model.

```
ISceneModel *CreateSM(const SFString& name, const SFString& DeviceDirectory
= "", const SFString& FirstSceneGraphName = "", const SFString&
FirstSceneIndexName = "")
```

Create a new scene model with the name specified by the input parameter name. The input parameter DeviceDirectory specifies the directory of scene storage. The input parameters FirstSceneGraphName and FirstSceneIndexName give the names of the first scene graph and the first spatial index respectively.

```
void CloseSM()
```

Close the scene model.

```
const ISceneModel*GetSceneModel() const
ISceneModel*GetSceneModel()
```

Retrieve the ISceneModel pointer pointing to the aggregated CSceneModelImp1 instance. It provides both constant or non-constant verisons.

3.10.1.2 Transaction Related Functions

```
void BeginTransaction()
SFBool RollbackTransaction()
SFBool CommitTransaction()
```

The above functions are used to begin, roll back and commit a transaction. See the introduction about transaction related functions in subsection 3.9.8 “ISceneStorage”.

3.10.1.3 Entity Related Functions

```
IEntity *CreateEntity(SFInt32 type, const SFString& name = "")
SFBool SaveEntity(SFID id)
SFBool RenameEntity(SFID ID, const SFString& Name)
Entity* FindEntity(const SFString& Name) const
SFString FindEntityName(SFID ID) const
```

These functions are the same as those in ISceneStorage. See the introduction about the entity related functions in subsection 3.9.8 “ISceneStorage”.

3.10.1.4 Scene Graph Related Functions

```
CSceneGraph*CreateSG(const SFString& name)
SFInt GetSGCount() const
CSceneGraph*GetSG(SFInt i)
```

The above functions are the same as those in ISceneModel. See the introduction about the scene graph related functions in subsection 3.9.7 “ISceneModel”.

```
SFBool DeleteSG(SFInt index)
```

It is the same as ISceneModel:: ForceDeleteGraph(index). See the introduction about the scene group related functions in subsection 3.9.7 “ISceneModel”.

```
SFBool AddNodeToSG(SFID node_id, SFID parent_id)
SFBool AddNodeToSG(ISGNode*pNode, CSGComposite*pParent)
```

The above two functions must be called in a certain transaction.

The first function is to make the scene graph node whose identifier is node_id be the child of another scene graph node whose identifier is parent_id. The second one does the same thing except the input parameters are scene graph node pointers.

A scene graph is a tree-like structure. Middle nodes have one or many children and one parent. The root node has one or many children but no parent. Leaf nodes have no children and one parent.

By using AddNodeToSG, developers can establish the child-parent relationship between two nodes.

```
SFID GetSGRoot(CSceneGraph *pSceneGraph) const
```

This function returns the root node identifier of the specified scene graph. According to the implementation of CSceneGraph, the root node of a scene graph is just the scene graph entity itself. Thus, this function just returns the identifier of the scene graph.

3.10.1.5 Spatial Index Related Functions

```
CKD2TreeIndex * CreateSI(const SFString& name);
SFInt GetSICount() const;
CKD2TreeIndex *GetSI(SFInt index);
```

The above functions are the same as those in ISceneModel. See the introduction about the spatial index related functions in subsection 3.9.7 “ISceneModel”.

```
SFBool DeleteSI(SFInt index);
```

It is the same as ISceneModel:: ForceDeleteIndex(index). See the introduction about the spatial index related functions in subsection 3.9.7 “ISceneModel”.

```
SFID GetSIRoot(CKD2TreeIndex * pSpatialIndex) const;
```

This function returns the root node identifier of the specified spatial index. According to the implementation of CKD2TreeIndex, the root node of a spatial index is just the spatial index entity itself. Thus, this function just returns the identifier of the spatial index.

3.10.1.6 ID Related Functions

```
IEntity* Pointer(SFID id) const;
```

This function translates the entity identifier to its pointer. If the entity has not been located in the host memory yet, there is no corresponding pointer and it returns NULL.

```
SFID ID(IEntity *p) const;
```

Return the entity identifier according to its pointer.

3.10.1.7 Garbage Collection Function

```
SFBool DeleteUnrefEntities();
```

It is the same as ISceneModel::DeleteUnrefEntities(). See the introduction of the other function in subsection 3.9.7 “ISceneModel”.

3.10.2 Usage of Scene Model Manipulator

There is a scene graph after creating a scene model in default. Typically, the code to create a scene model via a manipulator is listed as follows.

```
    ...
    CSceneModelManipulator2 Mpr;
    ISceneGraph * pSG;

// Create a new scene model.
      if(NULL == Mpr.CreateSM("my_scene"))
      return;

// Get first scene graph and the root node identifier.
      pSG = GetSG(0);
      SFID sg_root_id = Mpr.GetSGRoot(pSG);

// Begin a transaction.
      Mpr.BeginTransaction();

// Create a middle scene graph node.
      IFeature*pComposition=Mpr.CreateEntity(EnType_CSGComposition);

// Attach the middle node to the scene graph root as a child.
      Mpr.AddNodeToSG(Mpr.ID(pComposition),sg_root_id);

// Save the root, since the children field of the root has been changed.
      Mpr.SaveEntity(sg_root_id);
```

```
// Note:
// Even if we don't save the root node, the codes are also right.
// The reason is that AddNodeToSG has already called SaveEntity for us.

// Create a leaf scene graph node.
      IEntity*pSGLeaf = Mpr.CreateEntity(EnType_CSGLeaf);

// Attach the leaf node to the newly created middle node as a child.
      pComposition->AddChildren(TAddress<ISGNode>(pSGLeaf,
      pSGLeaf->GetID()));
// Note:
// pSGLeaf->GetID() is equivalent to Mpr.ID(pSGLeaf).

// Save the middle node, since the children field of the middle node
   has been changed.
      Mpr.SaveEntity(Mpr.ID(pComposition));
// Note:
// In fact, this above statement could be avoided without any influence.
// The reason is that the middle node is also created in the same
   transaction, and the scene model
// manipulator knows it must be saved.

// Commit the transaction.
      Mpr.CommitTransaction();

// Close the current scene model.
      Mpr.CloseSM();
      ...
```

3.11 Traversing Scene Model

As a rendering system, the Visionix scene model needs to be traversed and rendered per each frame. According to the organization of the Visionix scene model, traversing a scene model is equivalent to traversing scene graphs or spatial indices.

The Visionix system provides two means of traversing the scene model, such as iterator and visitor. These two means correspond to the design patterns, iterator and visitor.

3.11.1 Traverse via Iterator

As a design pattern, the iterator provides a way for users to visit elements of a container in a certain order without the need to know the details of the container's inside structure. Based on iterators, developers can easily write generic algorithms, which use containers but are independent of concrete containers. The typical

examples are the iterators defined in the C++ Standard Template Library.

For the Visionix scene model, such a container is either a scene graph or spatial index.

3.11.1.1 ISceneIterator

The Visionix system has defined an interface class ISceneIterator for various iterators of a scene graph and spatial indices. The class hierarchy of scene iterators is shown in Fig. 3.27.

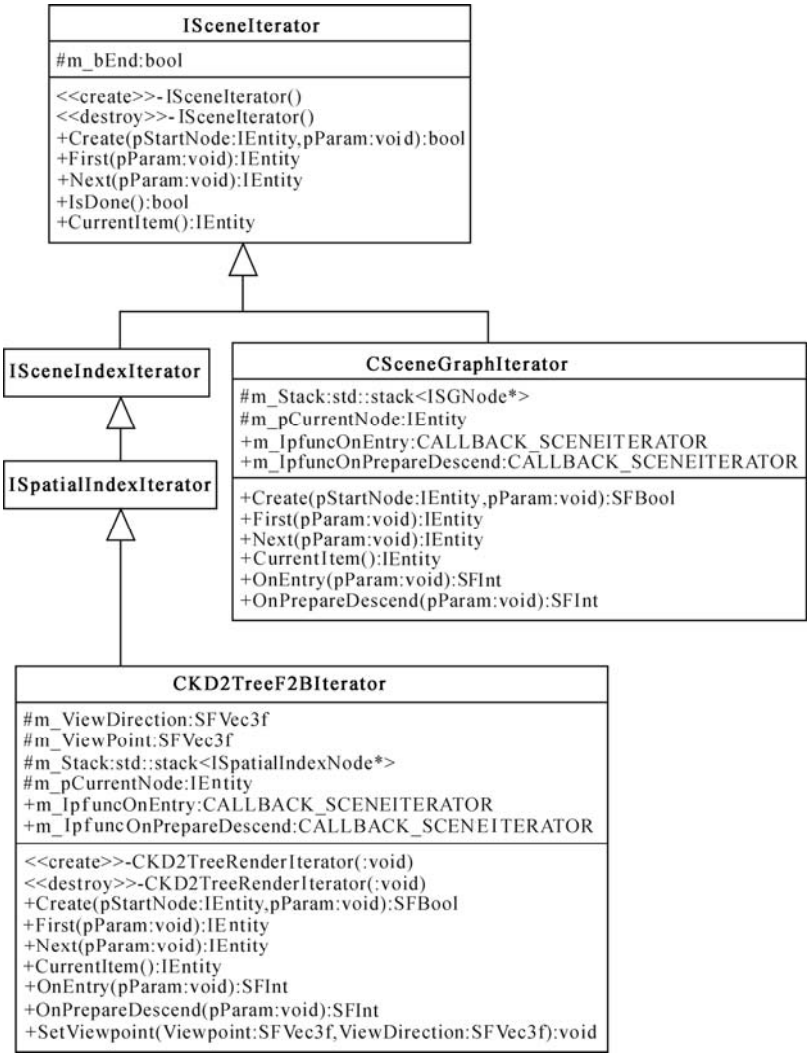

Fig. 3.27 The class hierarchy for scene iterators

The declaration of ISceneIterator is as follows:

```
typedef int (*CALLBACK_SCENEITERATOR)(ISceneIterator* pIterator,
void* pParam);
class ISceneIterator
{
public:
    ISceneIterator();
    virtual ~ISceneIterator(){};

    virtual bool Create(IFeature* pStartNode, void* pParam = NULL) = 0;
    virtual IEntity* First(void* pParam = NULL) = 0;
    virtual IEntity* Next(void* pParam = NULL) = 0;
    virtual bool IsDone() ;

    virtual IEntity* CurrentItem() = 0;

protected:
    bool m_bEnd;
};
```

Using terms in the C++ Standard Template Library (STL), ISceneIterator is a forward iterator. ISceneIterator::Next() can be used to make forward iteration. Nevertheless, ISceneIterator is more independent than those in STL.

The connection between an ISceneIterator instance and a target container is established through ISceneItrator::Create. The first parameter of ISceneItrator:: Create, pStartNode, specifies the starting node. ISceneIterator::First is to let the inner pointer point to the starting node and return the pointer. Joint use of Creat and First is equivalent to the statement

iterator iter = container.begin()

in STL. The method adopted by ISceneIterator avoids the container providing begin() or end() functions, furthermore makes containers have no dependency on ISceneIterator, and only ISceneIterator depends on containers. Similarly, ISceneIterator::IsDone checks whether the iterator has visited all elements by the iterator itself.

The member function CurrentItem returns the current node pointed by the inner pointer. The function Next returns the node after one forward step. It is similar to iterator++ in STL. Moreover, it is always true as Next() == CurrentItem(). If Next() returns NULL, it means the iteration should be terminated. At that time, IsDone returns true.

3.11.1.2 Iterator for Scene Graph

Class CSceneGraphIterator realizes the depth-first traversal of the scene graph. It merely visits the nodes which have ISNode interface. In other words, it does not

visit features in m_FeatureSet of any CSGLeaf instances.

The declaration of CSceneGraphIterator is listed as follows:

```
typedef int (*CALLBACK_SCENEITERATOR)(ISceneIterator * pIterator,
void * pParam);
class CSceneGraphIterator : public ISceneIterator
{
public:
    virtual bool Create(IEntity* pStartNode, void *pParam = NULL);
    virtual IEntity* First(void *pParam = NULL);
    virtual IEntity* Next(void *pParam = NULL);
    virtual IEntity* CurrentItem();

    virtual int OnEntry(void *pParam);
    virtual int OnPrepareDescend(void *pParam);

    CALLBACK_SCENEITERATOR m_lpfuncOnEntry;
    CALLBACK_SCENEITERATOR m_lpfuncOnPrepareDescend;
protected:
    IEntity * m_pCurrentItem; // It points to the current node.
    std::stack<ISGNode*>     m_Stack;
// The other implentation relationship member properties are omitted.
};
```

CSceneGraphIterator overrides the member function Create, First, Next and CurrentItem to provide implementations for them. The data member m_pCurrentItem points to the current visited node, and m_Stack is a stack to realize the depth first traverse.

Besides, there are two important virtual functions, OnEntry and OnPrepareDescend. Both of them are invoked at a certain time during traversing. These two functions are called event handlers, virtual function type(d) event handlers. Since the traverse is implemented via calling function Next continuously, the event handlers are invoked in function Next according to a certain order. The function First also has an opportunity to invoke the event handlers.

Developers can define CSceneGraphIterator's subclasses and override the event handlers to process scene graph nodes for their particular purposes.

When either of the event handlers is invoked, CSceneGraphIterator passes a pointer, void * pParam, to the event handler. This pointer is just the input parameter pParam of function Create, First or Next. If the input parameters pParam of Create, First and Next point to different objects respectively, the pointer pParam passed to the event handler points to the object pointed by the input parameter pParam of the last called function.

Corresponding to the event handlers, CSceneGraphIterator also defines two callback functions. In default, OnEntry and OnPrepareDescend are implemented by calling the corresponding callback functions. For example,

```
int CSceneGraphIterator::OnEntry(void *pParam)
{
    if(m_lpfuncOnEntry)
        return (*m_lpfuncOnEntry)(this, pParam);
    else
        return 1;
}
```

We regard the callback functions as another type of event handler. They are called callback function type(d) event handlers. Obviously, the callback function type(d) event handlers are more flexible than the virtual function type(d) event handlers. To provide a virtual function type(d) event handler, developers have to define a new subclass. Moreover, all instances of the new subclass have to share the same virtual function type(d) event handlers. On the other hand, to use callback function type(d) event handlers, what developers need to do is to write callback functions and assign the callback function pointers to a CSceneGraphIterator instance. Apparently, a different CSceneGraphIterator instance can have its own callback functions.

The statement for the type defining the callback function is as follows:
typedef int (*CALLBACK_SCENEITERATOR)(ISceneIterator * pIterator, void * pParam);

The callback function has two input parameters. The first one is a pointer pointing to the ISceneIterator instance, which owns this callback function as its event handler. The other one is the same as the input parameter of the virtual function type(d) event handler.

In the implementation of the function Next and First, only the virtual function type(d) event handlers are called. Then, the virtual function type(d) event handlers call their corresponding callback function type(d) event handlers respectively. Thus, the execution priority of the virtual function type(d) event handler is greater than the callback function type(d) one. We assume that someone defined a subclass of CSceneGraphIterator, overrode the virtual function OnEntry, and did not call any callback functions in the overridden function OnEntry. Then, the corresponding callback function type(d) event handler would not be called any more.

That means it is better to use only one type of event handler for one iterator. Since the two types of event handlers are almost the same in usage, we do not distinguish between them in the remaining introduction. We simply call them event handlers.

When a middle node is being visited, the corresponding event handlers are invoked in the following order.

OnEntry is invoked at the time when the node is just entered. If OnEntry returns 0, the visit of the node will be terminated, and all its child nodes will be skipped too. The next to-be-visited node is its sibling node.

OnPrepareDescend is invoked just before the iterator is going to visit the child nodes of the current node. If OnPrepareDescend returns 0, no child node

will be visited.

When a leaf node is visited, only OnEntry is invoked.

Fig. 3.28 shows an example of a scene graph. We use this example to explain how the event handlers are invoked during the traverse of the scene graph. The following codes illustrate how to traverse a scene graph through a CSceneGraphIterator instance:

```
CSceneGraphIterator iter;
iter.Create(&Node1);
iter.First();
while(!iter.IsDone()){iter.Next();}
```

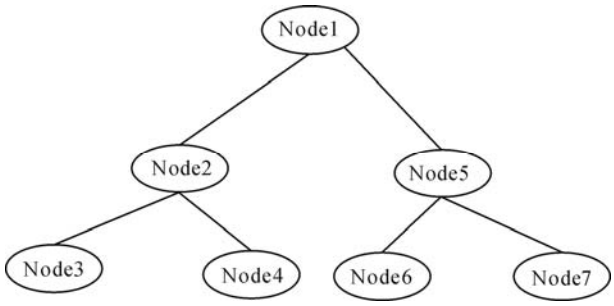

Fig. 3.28 An example of a scene graph

When the above codes are executed, the call case for event handlers is listed as follows:

(1) After calling iter.First(), OnEntry is invoked and iter.m_pCurrentItem points to Node1. This OnEntry is called Node1's OnEntry.

(2) Afterwards, iter.Next() is called and iter goes to Node2 from Node1. Since Node1 is a middle node, Node1's OnPrepareDescend will be invoked before iter.m_pCurrentItem is changed to be the address of Node2. Then, the return value of Node1's OnPrepareDescend will be examined.

- If the return value is 1, push Node5 and Node2 into ite.m_Stack, and keep Node2 on top. Then, pop Node2 from the stack, and let iter.m_pCurrentIterm point to Node2. Next, invoke OnEntry. Note, this OnEntry has been Node2's OnEntry.
- If the return value is 0, since iter.m_Stack is empty, the traverse should be terminated. Therefore, IsDone() will return true.

(3) We assume Node1's OnPrepareDescend returns 1. Then, Next() will be invoked again. At this time, iter.m_pCurrentItem points to Node2. Since Node2 is a middle node, Node2's OnPrepareDescend will be invoked.

- If OnPrepareDescend returns 1, push Node4 and Node3 into iter.m_Stack and keep Node3 on top. Afterwards, pop Node3 from the stack and let iter.m_pCurrentItem point to Node3. Then, invoke Node3's OnEntry.
- If OnPrepareDescend returns 0, pop Node5 from the stack and let

iter.m_pCurrentItem point to Node5. Then, invoke Node5's OnEntry.

(4) We assume Node2's OnPrepareDescend returns 1. Then, Next() will be invoked for the third time. Pop Node4 from iter.m_Stack, and let iter.m_pCurrentItem point to Node4. Then, invoke Node4's OnEntry.

(5) Keep doing so in the following.

In the following, we will use an example to illustrate the usage of CSceneGraphIterator.

```
struct CMyParam
{
    int count;
}
int MyCallback (ISceneIterator * pIterator, void * pParam)
{
    CSGComposition *p = dynamic_cast<CSGComposition*>
    (pIterator->CurrentItem());
    if(NULL != p && p->GetChildren().size() >= 3)
    {
        ((CMyParam *)pParam)->count++;
        return 1;
    }
    return 0;
}

...
CMyParam param;
param.count = 0;
CSceneGraphIterator iter;
iter.Create(pSG, &param); // pSG points to a scene graph object.
iter.m_lpfuncOnPrepareDescend = MyCallback;
iter.First();
while(!(iter.IsDone()))
{
    CSGLeaf * pL = dynamic_cast<CSGLeaf*>(iter.CurrentItem());
    if(NULL != pL)
        ProcessSGLeaf(pL);
    iter.Next();
}
...
```

The above code fragment shows how to traverse a sub-scene graph and process some nodes for particular purposes via a CSceneGraphIterator. In this example, any middle node which has at least three children is visited and counted. The variable param.count records the count. Moreover, the leaf nodes of these middle nodes are processed by the function ProcessSGLeaf.

3.11.1.3 Iterator for Spatial Index

Taking account of the extension of the other spatial indices, we introduce the abstract class ISceneIndexIterator, which acts as the common interface of various iterators for the spatial index.

Similar to CSceneGraphIterator, CKD2TreeIterator is a depth first iterator for a KD-tree, or say a CKD2TreeIndex instance. The event handlers of CKD2TreeIterator are the same as those of CSceneGraphIterator.

The CKD2TreeF2BIterator-typed iterator is able to visit KD-tree nodes in front-to-back order with respect to certain viewing parameters. When a middle node is visited, the iterator will sort its child nodes according to the spatial relationship between these nodes and the viewpoint. The node which is closest to the viewpoint along the viewing direction will be visited first. CKD2TreeF2BIterator has two event handlers, OnPrepareDescend and OnEntry. Its declaration is listed as follows:

```
class CKD2TreeF2BIterator : public ISpatialIndexIterator
{
public:
    virtual bool Create(IEntity* pStartNode, void *pParam = NULL);
    virtual IEntity* First(void *pParam = NULL);
    virtual IEntity* Next(void *pParam = NULL);
    virtual IEntity* CurrentItem();

    virtual int OnEntry(void *pParam);
    virtual int OnPrepareDescend(void *pParam);

    void SetViewpoint(const SFVec3f& Viewpoint, const SFVec3f&
        ViewDirection);

    CALLBACK_SCENEITERATOR m_lpfuncOnEntry;
    CALLBACK_SCENEITERATOR m_lpfuncOnPrepareDescend;
protected:
    IEntity * m_pCurrentItem; // It points to the current node.
    std::stack<ISpatialIndexNode*>       m_Stack;
    SFVec3f   m_Viewpoint;
    SFVec3f   m_ViewDirection;
}
```

The event handlers are the same as those in CSceneGraphIterator, so we will not discuss them any more.

Since any CKD2TreeComposite-typed node has just two child nodes, left child and right child, after the viewpoint and viewing direction are specified by the member function SetViewpoint, the iterator can guarantee to traverse the KD-tree nodes in front-to-back order with respect to the viewing parameters.

The front-to-back order is specified by CKD2TreeComposite::m_CutAxis,

CKD2TreeComposite::m_CutPosition, and the viewing parameters. Give a middle node N, which has two children C_{left} and C_{right}, the rules to determine whether the child is at front or back are listed as follows:

(1) If N.m_CutPositon < m_Viewpoint.v[N.m_CutAxis], C_{right} is at the front, C_{left} is at the back;

(2) If N.m_CutPositon > m_Viewpoint.v[N.m_CutAxis], C_{left} is at the front, C_{right} is at the back;

(3) If N.m_CutPositon == m_Viewpoint.v[N.m_CutAxis], then

- if ViewDirection.v[N.m_CutAxis]<0, Cleft is at the front, Cright is at the back;
- if ViewDirection.v[N.m_CutAxis]==0, Cright is at the front, Cleft is at the back.

Fig. 3.29 visualizes a two-dimensional KD-tree, which is the result of 3 times space cutting. The entire region is denoted by R_0. After cutting R_0 along the *x*-axis, two regions R_1 and R_2 are formed and we have R_0.m_CutAxis = *X*_Axis {= 0}, R_0.m_CutPosition=100. The two regions R_3 and R_4 are the cutting results of R_2, and we have R_2.m_CutAxis = *Y*_Axis{=1}, R_2.m_CutPosition=50. The two regions R_5 and R_6 are the cutting results of R_4 and we have R_4.m_CutAxis = *X*_Axis, R_4.m_CutPosition = 50. The viewpoint is highlighted by the red circle and the viewing direction is illustrated by the red arrow.

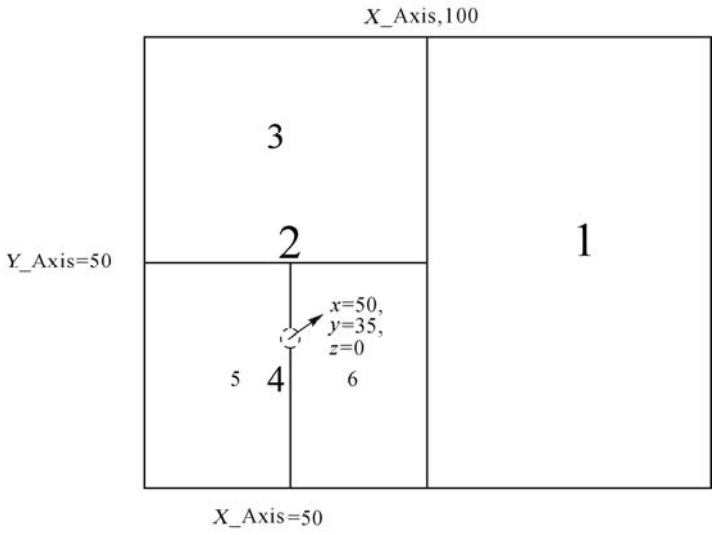

Fig. 3.29 An example of a KD-tree

In this example, according to the above rules, the traverse order is R_0, R_2, R_4, R_6, R_5, R_3, R_1 if the starting node is R_0.

The following codes process leaf nodes in front-to-back order. They are very useful for many algorithms:

```
CKD2TreeF2BIterator iter;
iter.Create(&theKDTree);
iter.First();
while(!iter.IsDone())
{
    CSpatialIndexLeaf * p;
    p=dynamic_cast<CSpatialIndexLeaf*>(iter.CurrentItem());
    if(NULL != p)
    {
        ProcessLeaf(p);
    }
    iter.Next();
}
```

3.11.2 Traverse via Visitor

In fact, the visitor encapsulates a set of operations, each of which is supposed to handle a class of elements in a certain container. To perform the traverse of a container, these encapsulated operations should have some codes to perform navigation from one element to another.

3.11.2.1 IVisitor

The Visionix system defines the abstract class IVisitor for various visitors which are able to access all kinds of entities. The base class IEntity is able to accept an IVisitor instance.

```
class IEntity
{
...
public:
    virtual void Accept(IVisitor * pVisitor) = 0
};

class IVisitor
{
public:
    virtual void Apply(CSceneGraph * pEntity) = 0;
    virtual void Apply(CSGLeaf * pEntity) = 0;
    virtual void Apply(CSGComposite * pEntity) = 0;
    virtual void Apply(CSGLayer * pEntity) = 0;
    virtual void Apply(CKD2TreeIndex * pEntity) = 0;
```

```
    virtual void Apply(CKD2TreeComposite * pEntity) = 0;
    virtual void Apply(CSpatialIndexLeaf * pEntity) = 0;
    virtual void Apply(CRelation_SISG * pEntity) = 0;
    virtual void Apply(CViewpoint * pEntity) = 0;
    virtual void Apply(CSpotLight * pEntity) = 0;
    …
}
```

IVisitor provides an abstract virtual function Apply(ClassType*) for each entity class. The concrete subclass of IVisitor should implement these abstract virtual functions. The input parameter pEntity points to the currently visited entity by the visitor.

The number of the Apply functions is equal to that of the concrete subclasses of IEntity, so as to assure the visitor access any entity. Therefore, defining a concrete IVisitor class requires writing a lot of codes. It is the main drawback of visitor. In practice, we always define a new concrete visitor based on an existing concrete visitor to reduce code writing.

Every concrete entity class should provide the implementation of the abstract virtual function IEntity::Accept. Visionix provides the default implementations, as the following code shows.

```
void CSceneGraph::Accept(IVisitor* pVisitor)
{
     pVisitor->Apply(this);
}
void CSGLeaf::Accept(IVisitor* pVisitor)
{
     pVisitor->Apply(this);
}
void CSGComposite::Accept(IVisitor* pVisitor)
{
     pVisitor->Apply(this);
}
void CSGLayer::Accept(IVisitor* pVisitor)
{
     pVisitor->Apply(this);
}
…
```

It is rather simple to implement the Accept function of each concrete entity class by calling pVisitor->Apply(this). This guarantees the codes of each entity class have no idea of how the entities are processed, and only depend on the interface IVisitor.

3.11.2.2 CBasicVisitor

For visiting a scene graph or spatial index, there are no traverse related codes in IVisitor. In other words, there is no place in IVisitor to specify the traverse order and the traverse range. Thus, to achieve the traverse of a scene graph or spatial index, the codes which carefully handle the middle nodes of either scene graph or spatial index must be written in any concrete subclass of IVisitor. For the scene graph visitor, the codes should be in the Apply functions for CSGComposition and its subclasses. For the spatial index visitor, the codes should be in the Apply functions for CSpatialIndexComposition and its subclasses.

To reduce the workload of the writing visitor, the Visionix system provides CBasicVisitor, a basic implementation of IVisitor. CBasicVisitor is able to traverse both scene graph and spatial index.

CBasicVisitor realizes a depth first traverse. It takes the following measures for middle nodes in tree structures. Here, we show an example of how to handle a CSGComposition node. The others are almost the same.

```
void CBasicVisitor::Apply(CSGComposition * pEntity)
{
     if(!OnEntry(pEntity))
        return;
     if(OnPrepareDescend(pEntity)){
        for(each child node pointer pChild in pEntity->GetChildren()){
            pChild ->Accept(this);
        }
    }
     OnAscend(pEntity);
}
```

For the leaf node classes in the scene graph and spatial index, such as CSGLeaf, CSGDynLeaf and CSpatialIndexLeaf, CBasicVisitor provides corresponding Apply functions, as listed in the following:

```
void CBasicVisitor::Apply(CSGLeaf* pEntity)
{
     OnEntry(pEntity);
}

void CBasicVisitor::Apply(CSGDynLeaf* pEntity)
{
     OnEntry(pEntity);
}

void CBasicVisitor::Apply(CSpatialIndexLeaf* pEntity)
{
     OnEntry(pEntity);
}
```

Like iterators, CBasicVisitor has event handlers, i.e., OnEntry in the above codes, to make extensions. Of course, we also introduce the callback function type(d) event handlers. The callback function prototype is defined as typedef int (*CALLBACK_VISITOR)(IVisitor*pVisitor, IEntity*pEntity, void*pParam).

The declaration of CBasicVisitor is listed as follows:

```
class CBasicVisitor : public class IVisitor
{
public:
    CBasicVisitor();
    virtual ~CBasicVisitor();

    void SetParam(void * pParam){m_pParam = pParam;}
    void* GetParam(){return m_pParam;}
    // Event handlers
    virtual int OnEntry(IEntity * pEntity);
    virtual int OnPrepareDescend(IEntity * pEntity);
    virtual int OnAscend (IEntity * pEntity);

// Event functions
    CALLBACK_VISITOR m_lpfnOnEntry;
    CALLBACK_VISITOR m_lpfnOnPrepareDescend;
    CALLBACK_VISITOR m_lpfnOnAscend;

// The implementations of Apply interfaces are omitted.
    …
protected:
    void * m_pParam;
}
```

The virtual function type(d) event handler is implemented by using the corresponding callback function. The following is an example:

```
int CBasicVisitor::OnEntry(IEntity * pEntity)
{
    if(m_lpfnOnEntry)
        return (*m_lpfnOnEntry)(this, pEntity, m_pParam);
    else
        return 1;
}
```

Observed from the implementation of Apply functions, the return value of OnEntry is used to specify whether the visitor continues to visit this node and its children. The return value of OnPrepareDescend is used to specify whether all children are visited and OnAscend is invoked after all child nodes and contained entities have been visited.

Developers can realize their own visitor through providing their event handlers.

3.11.2.3 CBoundedFeatureVisitor

CShapeVisitor is used to visit the features, which are of IBoundedObject and IFeature class, such as the subclasses of IShapeFeature, IGroupingFeature and IAnimatedFeature.

In the Visionix scene model, besides the scene graph and spatial index, some feature also has a hierarchy, namely a tree representation. In this kind of tree representation, the middle node types include CGroup, CTransformGroup, CStaticLOD, CComposeShape and CSuperShape, and the leaf node types include CShape, CSkinAnim and CBillboard.

```
class CBoundedFeatureVisitor
:public IVisitor
{
public:
// The following functions handle intermedia nodes
    virtual int Apply(CSuperShape*);
    virtual int Apply(CComposedShape*);
    virtual int Apply(CTransformGroup*);
    virtual int Apply(CStaticLOD*);
    virtual int Apply(CGroup*);
// The following functions handle leaf nodes.
    virtual int Apply(CShape*);
    virtual int Apply(CBillboard*);
    virtual int Apply(CSkinAmin*);

    virtual int OnEntry (IEntity* pEntity);
    virtual int OnPrepareDescend(IEntity* pEntity);
    virtual int OnAscend (IEntity* pEntity);
    CALLBACK_VISITOR m_lpfnOnEntry;
    CALLBACK_VISITOR m_lpfnOnPrepareDescend;
    CALLBACK_VISITOR m_lpfnOnAscend;

    void SetParam(void* pParam){m_pParam = pParam;}
    void* GetParam(){return m_pParam;}
//We provide dummy implementation for the other Apply interface functions.
    ...
protected:
    void * m_pParam;
}
```

The event handlers are the same as that of CBasicVisitor.

Here, we give two examples to show how CBoundedFeatureVisitor handles the middle node and leaf node.

```
void CBoundedFeatureVisitor::Apply(CSuperShape* pEntity)
```

```
{
    if(!OnEntry(pEntity))
        return;
    if(OnPrepareDescend(pEntity)){
        for(each subshape pointer pSubshape in pEntity->GetSubshapes()){
            pSubshape ->Accept(this);
        }
    }
    OnAscend(pEntity);
}

void CBoundedFeatureVisitor::Apply(CShape* pEntity)
{
    OnEntry(pEntity);
}
```

3.11.2.4 Customized Visitor

Based on CBasicVisitor and CBoundedFeatureVisitor, and by using event handlers, developers can easily customize a visitor for their own purpose.

Suppose we need a visitor which is able to traverse a scene graph, visit all bounded features and compute its bounding box.

Note that the scene graph node has no IBoundedObject interface. Thus we insert a bounding box as an attribute to the node's attribute set.

The implementation principle is to use CBasicVisitor and CBoundedFeatureVisitor in cascade. That is, we use CBasicVisitor to traverse the scene graph and use CBoundedFeatureVisitor to traverse the features contained by each leaf node. Thus, we need to provide the event handlers, OnEntry and OnAscend, for both CBasicVisitor and CBoundedFeatureVisitor.

In a tree structure, the bounding box of a parent node relies on the bounding boxes of its descendant. Hence, the bounding box of a middle node is computed by its OnAscend event handler. The corresponding callback functions are

```
callback_ AscendSGComposition(...)// for CBasicVisitor,
callback_AscendGroup (...)        // for CBoundedFeatureVisitor.
```

The event handler CBasicVisitor::OnEntry is based on the callback function callback_EntrySGLeaf. The function firstly checks whether the input node is a leaf node. If it is a leaf node, CBoundedFeatureVisitor is used to visit the features of this leaf node and compute the bounding boxes, as the bold-font statements show in the following codes. After processing all features, all features' bounding boxes are combined to yield the bounding box of the leaf node.

```
main()
{
    ...
    CBasicVisitor visitor;
    visitor.m_lpfnOnEntry = callback_EntrySGLeaf;
    visitor.m_lpfnOnAscend = callback_AscendSGComposition;
    visitor.Apply(pSceneGraph); // pSceneGraph points to a CSceneGraph
    object.
...}

int callback_EntrySGLeaf(IVisitor*pVisitor, IEntity*pEntity, void*pParam)
{
    CSGLeaf * p = dynamic_cast<CSGLeaf*>(pEntity);
    if(p){
        CBoundedFeatureVisitor BFVisitor;
        BFVisitor.m_lpfnOnEntry(callback_EntryShape);
        BFVisitor.m_lpfnOnAscend(callback_AscendGroup);
        for(i=0; i< p->GetFeatureSet().size(); ++i)
        {
            // The bounding box of f will be computed during the visit.
            p->GetRelation()[i]->Accept(&BFVisitor);
        }
        Compute the bounding box of the leaf node pointed by p according
        to the bounding boxes of all features in p->GetFeatureSet().
    }
    return 1;
}

int callback_AscendSGComposition(IVisitor*pVisitor, IEntity*pEntity,
void * pParam)
{
    CSGComposite * p = dynamic_cast< CSGComposite *>(pEntity);
    if(p){
        Compute the bounding box of the intermediate node pointed
        by p according to the bounding boxes of all children in
        p->GetChildren().
    }
    return 1;
}

int callback_EntryShape(IVisitor*pVisitor, IEntity*pEntity, void*pParam)
{
    swtich(p->EntityType())
    {
    case EnType_CShape:
        Compute the bounding box of the shape pointed by pEntity;
        break;
    case EnType_CBillboard:
        Compute the bounding box of the billboard pointed by pEntity;
```

```
            break;
        case EnType_CSkinAnim:
            Compute the bounding box of the skin animation object pointed
            by pEntity;
            break;
        }
        return 1;
}

int callback_AscendGroup(IVisitor*pVisitor,IEntity*pEntity,void*pParam)
{
        switch(p->EntityType())
        {
        case EnType_CSuperShape:
            Compute the bounding box of the CSuperShape object pointed
            by pEntity according to the bounding boxes of all subshapes.
            break;
        case EnType_ CComposedShape:
            Compute the bounding box of the CComposedShape object pointed
            by pEntity according to the bounding boxes of all children.
            break;
        case EnType_ CTransformGroup:
        case EnType_CStaticLOD:
        case EnType_CGroup:
            Compute the bounding box of the IGroupingFeature object pointed
            by pEntity according to the bounding boxes of all children.
            break;
        }
        return 1;
}
```

3.11.2.5 Built-in Visitors

The Visionix system also provides several built-in visitors as follows:

(1) CCopySubTreeVisitor, which is for duplicating a sub-scene graph and updating the corresponding KD-tree. Its root is a specified node's child. All duplicated scene graph nodes are newly created objects, but the contained features are not created. Only the feature references are duplicated into the new scene graph nodes.

(2) CDeleteSubTreeVisitor, which is to delete a sub-scene graph, and update the corresponding KD-tree.

(3) CCollectSubTreeRelationVisitor, which is to collect all CRelation_SISG instances related to a sub-scene graph.

(4) CUpdateSubTreeAttributeVisitor, which is to update a certain attribute of the nodes in a sub-scene graph.

3.12 Rendering Engine

The Class CRenderingEngine is the core of the entire rendering system. It connects several parts of the rendering system. Moreover, it is also configurable to meet the requirements of various applications. Meanwhile, it is a façade of the rendering system, as it provides a set of APIs for developers.

In this section, we first introduce some member functions of CRenderingEngine, which can be regarded as some Application Programming Interfaces (API) from the viewpoint of developers, and give a simple example of how to use these member functions. Then we present the composition of the CRenderingEngine to reveal the organization of a CRenderingEngine object.

In sections 3.13 to 3.21, we give a detailed description of each rendering engine component. In Section 3.22, we present the implementation details of the rendering engine framework.

3.12.1 CRenderingEngine

3.12.1.1 The Important Member Functions of the CRenderingEngine

In the following, we present several important member functions of the CRenderingEngine:

```
class CRenderingEngine
{
…
public:
    int  Configure(const SFString& ConfigFileName);
    int  Initialize(const CWinInfo& WinInfo);
    int  OpenSceneModel(const SFString& SceneModelName, int
         ActiveSceneGraph=0);
    void CloseSceneModel();
    void SetRenderTraverseRoot(IEntity * pRoot);
    void SetCamera(const CCamera& Camera);
    void GetCamera(CCamera& Camera);
    void DoRendering();
    void SwapBuffer();
    void Finalize();
public:
// Overridables
    virtual int OnConfigure();
…
}
```

(1) Configure. The function Configure configures the rendering engine according to the config file, whose file name is the input parameter ConfigFileName. When the string ConfigFileName is empty or the file cannot be found, the function uses the default settings. This function should be called just after the CRenderingEngine object is constructed. OnConfigure as an overridable function is invoked in Configure, which gives developers the opportunity to configure the rendering engine in the desired way. If OnConfigure returns non-zero value, the default configuration is skipped.

(2) Initialize. The function Initialize initializes the rendering engine when the rendering output window is ready. It should not be invoked before invoking the function Configure. The input parameter is a WinInfo object, which specifies the parameters of the rendering output window. WinInfo depicts the window where rendering results are displayed. WinInfo's type is CWinInfo, which is a kind of wrap for a window defined in the operating system. For the version of the Visionix system which is based on Microsoft Windows, CWinIno is defined as

```
struct CWinInfo
{
HWND    hWnd;    // window handler
HGLRC   hRC;     // OpenGL context handler
}.
```

When Initialize is invoked, the output window must be created outside the CRenderingEngine, and its window handle is passed into the rendering engine through the WinInfo object. The OpenGL context can be created either outside or inside the engine. Namely, WinInfo.hRC can be a valid handle or NULL. If WinInfo.hRC is NULL, the function Initialize creates it according to default settings.

(3) SetCamera. The function updates the CRenderingEngine's internal camera with the input parameter Camera. The CRenderingEngine's internal camera is used in the function DoRendering. The input Camera is copied by value. The viewing parameters are represented by CCamera, as follows:

```
struct CCamera
{
    CRect           Viewport;
    CViewFrustum    ViewFrustum;
    SFVec3d         Viewpoint;
    SFVec3d         ViewTarget;
    SFVec3d         UpDirection;
    enum{PERSPECTIVE, ORTHOGONAL, ORTHOGONAL2D} ProjectionMode;
};

struct CRect
{
```

```
    SFInt32     Left, Right, Top, Bottom;
};
struct CViewFrustum
{
    SFDouble    Left, Right, Top, Bottom, Near, Far;
};
```

(4) GetCamera. The function copies the values of the internal camera to the input paramter.

(5) SetRenderTraverseRoot. If one scene graph or spatial index node, pointed by the parameter pEntity, is set to be the render traverse root by this function, all features related to its descendant nodes will be traversed and rendered.

(6) OpenSceneModel. The function OpenSceneModel is for opening a scene model whose name is given by the input parameter SceneModelName and for setting the to-be-rendered scene graph. The string SceneModelName can be the full path of the file, or just the file name. When the string SceneModelName is empty or the scene model cannot be found in the default path, the function will return 0. It should be called after calling Configure and Initialize. After executing OpenSceneModel, the first kd-tree is regarded as the default render traverse root. Moreover, according to the first kd-tree (or the scene graph corresponding to the kd-tree), the engine's internal camera is set to a set of default values. It means that after calling OpenSceneModel, developers can use GetCamera to retrieve the default viewing parameters.

(7) CloseSceneModel. The function closes the opened scene model. The default destruction function calls CloseSceneModel.

(8) DoRendering. The function does all rendering tasks to generate one image in the back frame buffer of the rendering output window, with respect to the internal camera. By default, it renders the entities in the active scene graph. For most walk-through application, the application developers should call SetCamera and DoRendering at a certain rate with the changing viewing parameters.

(9) SwapBuffer. The function swaps the back frame buffer to the front frame buffer. It is usually called just after calling DoRendering.

(10) Finalize(). The function stops the render engine, and releases the window related resouces allocated for the engine. When the engine is finalized, the states inside the engine are set back to the uninitialized but configured states. That is, after calling the function Finalize, the function Initialize should be called again, but there is no need to call the function Configure if the engine is to be reused again. The destructor calls this function automatically.

3.12.1.2 The Basic Example of Using the CRenderingEngine

The following pseudo codes show a typical example of using the CRenderingEngine. The codes are based on the framework of Microsoft Foundation Classes. The class CMyWnd can be used in a walkthrough application. The CMyWnd is able to open

a scene model and display the features in the first scene graph, and respond to mouse input with changing viewing parameters.

```
// CMyWnd.h
class CMyWnd : public CWnd
{
protected:
    CRenderingEngine m_RE;
    CCamera          m_Camera;

    DECLARE_MESSAGE_MAP()
    afx_msg int  OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnSize(UINT nType, int cx, int cy);
    afx_msg void OnPaint();
    afx_msg void OnFileOpen();
...
};

// CMyWnd.cpp
BEGIN_MESSAGE_MAP(CMyWnd, CWnd)
    ON_WM_CREATE()
    ON_COMMAND(ID_FILE_OPEN, OnFileOpen)
    ON_WM_SIZE()
    ON_WM_PAINT()
    ON_WM_MOUSEMOVE()
END_MESSAGE_MAP()
int CMyWindow::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if(!m_RE.Configure()) return false;
  CWinInfo wi;
  wi.hWnd = CreateWindow(...);
  ...
  if(!m_RE.Initialize(wi)) return false;
}

void CMyWindow::OnFileOpen ()
{
    CFileDialog dlg(TRUE, "xw", NULL, 0, "*.xw;*.wrl");
    if(dlg.DoModal() == IDOK){
        if(!m_RE.OpenSceneModel((LPCSTR)dlg.GetPathName()) return false;
        m_RE.GetCamera(m_Camera); // Get the default values
    }
}

void CMyWindow::OnSize(UINT nType, int cx, int cy)
{
    m_Camera.Viewport.Right = cx;
```

```
    m_Camera.Viewport.Top = cy;
    double ratio = (double)cy/(double)cx;
    m_Camera.ViewFrustum.Top= m_Camera.ViewFrustum.Bottom
        +ratio*(m_Camera.ViewFrustum.Right-m_Camera.ViewFrustum.Left);
    m_RE.SetCamera(m_Camera);
}

void CMyWindow::OnPaint()
{
    m_RE.SetCamera(m_Camera);
    m_RE.DoRendering();
    m_RE.SwapBuffer();
}

void CMyWindow::OnMouseMove(UINT nFlags,CPoint point)
{
    Modify m_Camera.Viewpoint, m_Camera.ViewTarget, m_Camera.UpDirection
    according to the mouse movement and flag.
    OnPaint();
}
```

3.12.2 The Composition of the CRenderingEngine

The main data members of the CRenderingEngine are listed as follows:

```
class CRenderingEngine
{
protected:
/* m_WinInfo represents the window and the OpenGL context for displaying
the rendering results. */
    CWinInfo                    m_WinInfo;

/* m_pSceneModelMpr points to a CSceneModelManipulator2 instance. As
introduced in the section 3.10, CSceneModelManipulator2 is a wrap of
the scene model. By default, a CSceneModelManipulator instance is
aggregated by the rendering engine. However, in the case where there
are many CRenderingEngine instances who share one scene model,
m_pSceneModelMpr can also be exploited as a reference to an external
CSceneModelManipulator2 instance. */
    CSceneModelManipulator2*    m_pSceneModelMpr;

/* m_GPUResourceMpr takes charge of creating, sharing and releasing
various resources in GPU memory. */
```

```
    CGPUResourceManager          m_GPUResourceMpr;

/* m_RenderQueueMgr manages multiple render queues. Each render queue
is a queue for holding various to-be-rendered objects. */
    CRenderQueueManager          m_RenderQueueMgr;

/* m_CameraMgr manages multiple cameras, each of which provides the
viewing parameter for a specific pre-render and a specific render
pipeline. */
    CCameraManager               m_CameraMgr;

/* m_RenderTargetMgr manages multiple render targets. Each render
target specifies a buffer that stores the rendering result of a render
pipeline. */
    CRenderTargetManager         m_RenderTargetMgr;


/*m_PreRenderMgr manages multiple pre-renders, or say IPreRender
objects, and performs multiple pre-rendering. One kind of pre-render
can perform a kind of pre-rendering. The main purpose of pre-rendering
is to yield render queues. */
    CPreRenderManager            m_PreRenderMgr;

/* m_RenderPipelineMgr manages multiple IRenderPipeline instances.
Each IRenderPipeline instance represents a rendering pipeline, which
is able to render the elements of several render queues onto a render
target. */
    CRenderPipelineManager       m_RenderPipelineMgr;

/* m_pRenderTraverseRoot is set by the function SetRenderTraverseRoot,
which is introduced in the section APIs of CRenderingEngine. It points
to a node either in the scene graph or in the spatial index. All descendant
nodes of this node will be traversed and the related features will
be rendered. */
    IEntity *                    m_pRenderTraverseRoot;

/* m_pStartPreRender points to the pre-render that will be firstly
performed in control flow. */
    IPreRender   *               m_pStartPreRender;
    ...
}
```

Fig. 3.30 shows the basic control flow and data flow inside a CRenderingEngine instance. The solid lines represent control flow and the dash lines represent data flow.

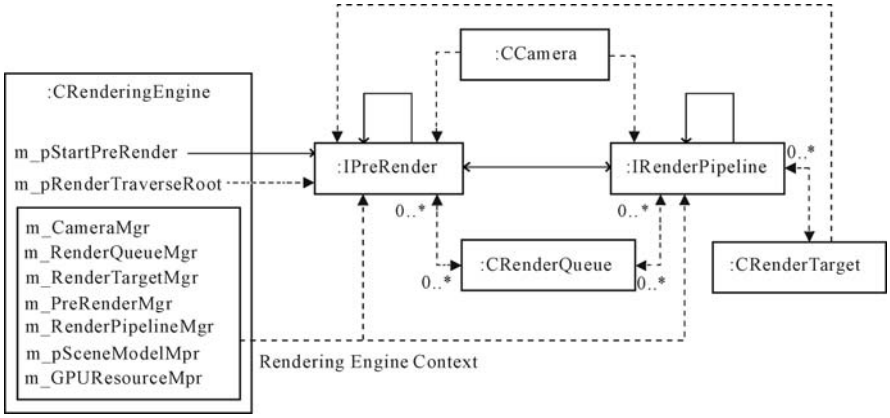

Fig. 3.30 The control flow and data flow inside a CRenderingEngine instance

In the CRenderingEngine, the data member m_pStartPreRender points to an IPreRender instance, whose life period is managed by m_PreRenderMgr. This IPreRender instance is the first pre-render pass of the whole rendering procedure. Both IPreRender and IRenderPipeline instances as control flow units can be connected to each other. Not only different IPreRender instances (or IRenderPipeline instances) can be connected, but also an IPreRender instance and an IRenderPipeline can be connected along the direction from IPreRender to IRenderPipeline, or along the contrary direction. For more information about the control flow, refer to Section 3.17 “Render Control Unit”. By connecting IPreRender and IRenderingPipeline instances, various control flows can be formed inside the rendering engine. In most cases, IPreRender and IRenderPipeline instances are managed by m_PreRenderMgr and m_RenderPipelineMgr respectively.

By default, m_pRenderTraverseRoot is passed to all IPreRender instances, and it is regarded as the first node to be visited of the scene traverse.

Besides, each IPreRender instance takes the following instances as inputs, such as a CCamera instance, multiple CRenderQueue instances, and a render target. Moreover, a rendering engine context is also passed into each IPreRender instance to provide global knowledge. The rendering engine context consists of the five managers and two manipulators defined in the CRenderingEngine. IPreRender instances use these inputs to perform some computation and output several CRenderQueue instances. The most typical computation of IPreRender is to determine and collect visible objects while traversing the scene.

Each IRenderPipeline instance takes the following instances as inputs, such as a CCamera instance, multiple CRenderQueue instances, and multiple CRenderTarget instances. The rendering engine context is also passed on. The output of a IRenderPipeline instance includes a CRenderTarget instance and multiple CRenderQueue instances. The main function of IRenderPipeline is to render elements of render queues onto the output render target.

By default, m_CameraMgr manages a prime camera, which collects the

viewing parameters for the final output window. Users of the CRenderingEngine can call CRenderingEngine::SetCamera to set the data of the prime camera.

As mentioned above, IPreRender and IRenderPipeline instances form the control flow. By providing various concrete subclasses of IPreRender and IRenderPipeline, and making right connections between these instances, CRenderingEngine instances can be configured and extended flexibly.

3.13 Render Queue and Its Manager

The class CRenderQueue specifies a render queue which consists of entities. In most cases, these entities are supposed to be rendered in the following render pipelines. The declaration is shown as follows:

```
struct CRenderQueueElement
{
    SFMatrix44f  * m_pMatrix;
    IFeature     * m_pFeature;
}

class CRenderQueue
{
public:
    TMF<CRenderQueueElement> m_Queue;
    SFBool                   m_bMatrixAggregated;
    SFInt                    m_Tag;
    IRenderQueueExtension *  m_pExtension;
    void PushBack(const CRenderQueueElement&);
    void PushFront(const CRenderQueueElement&);
    …
}
```

The data member m_Queue is the array of CRenderQueueElement instances. CRenderQueueElement::m_pMatrix points to a matrix that transforms the local coordinate frame of the object referenced by m_pFeature to a certain coordinate frame. In most cases it transforms the local coordinate frame to the world coordinate frame. CRenderQueue::m_bMatrixAggregated is a flag to indicate whether each CRenderQueueElement instance in m_Queue aggregates the matrix pointed by m_pMatrix. If the flag is true, the matrix is aggregated, i.e., it has the same lifetime as the host CRenderQueueElement instance.

The property m_Tag is used to indicate the queue's usage. The property m_pExtension is for user-defined data. IRenderQueueExtension is an abstract class.

The function PushBack is to append an IEntity instance to the end of m_Queue. The function PushFront is to insert an IEntity instance at the beginning of m_Queue.

The declaration of some important parts of CRenderQueueManager is listed as follows:

```
class CRenderQueueManager
{
public:
    CRenderQueue * Give(const SFString& QueueName);
    CRenderQueue * operator [](const SFString& QueueName);
    ...
protected:
    Map<SFString, CRenderQueue> m_RenderQueues;
};

CRenderQueue * CRenderQueueManager::Give(const SFString& QueueName)
{
    if m_RenderQueues[QueueName] exists, return its pointer;
    else{
        Insert a new render queue to the map m_RenderQueues with its
        key QueueName;
        return the new queue's pointer;
    }
}

CRenderQueue* CRenderQueueManager::operator [](const SFString& QueueName)
{
    if m_RenderQueues[QueueName] exists, return its pointer;
    else return NULL;
}
```

A CRenderQueueManager instance is a render queue container. Inside the container there is a map, the key of which is the unique name of the render queue. The overridden operator [](const SFString& QueueName) is used to search the corresponding render queue whose name is specified by the input parameter.

The function Give(const SFString& QueueName) can be used to produce a render queue for the outside. It should be noted that Give does not always create a new instance of CRenderQueue. It firstly checks whether the request render queue has ever been created. If so, the function finds the render queue from the map and gives the queue to the outside. If not, the function creates it.

3.14 Camera Manager

The declaration of CCamera can be referred to in subsection 3.12.1 “CRenderingEngine”.

Similar to CRenderQueueManager, a CCameraManager instance is a container for cameras, namely CCamera instances. There is a map inside, the key of which

is the camera's name and is unique in the whole map.

The overridden operator [](const SFString& CameraName) can be used to find the render queue by a specified name. The function Give(const SFString& CameraName) can be used to yield a named camera, which is either a newly created instance or an existing instance with the same name in the map.

```
class CCameraManager
{
public:
    CCamera * Give(const SFString& CameraName);
    CCamera * operator [](const SFString& CameraName);
    …
protected:
    Map<SFString, CCamera> m_Cameras;
};
```

3.15 GPU Resources and Its Manipulator

The Visionix system encapsulates several kinds of rendering resources, which can be directly allocated in the memory built-in graphics card. This kind of memory is called GPU memory. The ordinary host memory is called CPU memory to distinguish it from GPU memory. IGPUResource is the base class of these resources. The reference count is adopted to solve the sharing problem. That means, when one IGPUResource instance is shared by multiple objects, the IGPUResource instance can be released from GPU memory safely if, and only if, these objects who share this resource are all destroyed.

The approach to reference counting is that we set a reference count m_ReferenceCount in IGPUResource. The reference count records how many times the IGPUResource instance has been referred to.

In the default constructor of IGPUResource, m_ReferenceCount is set to be 1. After calling IGPUResource::Reference once, m_ReferenceCount is increased by 1. After calling IGPUResource::Unreference once, m_ReferenceCount is decreased by 1.

We define the function Reference, Unreference and destructor to be protected, and let CGPUResourceManipulator be IGPUResource's friend class. Only CGPUResourceManipulator can invoke these functions, so that CGPUResourceManipulator takes full charge of these resources' life period.

The declaration of IGPUResource is listed as follows:

```
class IGPUResource
{
public:
    friend class CGPUResourceManipulator;
    IGPUResource():m_ReferenceCount(1){}
```

```
    SFUInt GetReferenceCount();
    SFInt  GetGPUMemoryCost() const;
protected:
    IGPUResource* Reference(){++m_ReferenceCount;}
    void Unreference(){--m_ReferenceCount;}
    virtual ~ IGPUResource ();
protected:
    SFInt   m_ReferenceCount;
    SFInt   m_GPUMemoryCost;
}
```

The data member GPUMemoryCost records the size of GPU memory occupied by the resource.

So far, three categories of resources including texture resources, buffer resources and shader program have been defined:

3.15.1 Texture Resource

We have defined two kinds of texture resources:

```
class CImageTextureGPUResource:public IGPUResource
{
public:
    virtual void Bind() const
    {glBindTexture(GL_TEXTURE_2D,m_Handle);}
    virtual void Unbind() const
    {glBindTexture(GL_TEXTURE_2D,0);}
protected:
    GLuint      m_Handle;
};

class CCubeMapTextureGPUResource:public IGPUResource
{
public:
    virtual void Bind() const
    {glBindTexture(GL_TEXTURE_CUBE_MAP,m_Handle);}
    virtual void Unbind() const
    {glBindTexture(GL_TEXTURE_CUBE_MAP,0);}
protected:
    GLuint      m_Handle;
};
```

Both CImageTextureGPUResource and CCubeMapTextureGPUResource contain a handle, which specifies the corresponding GPU resource allocated by OpenGL context.

Function Bind is to make the texture object ready for use. Function Unbind is to remove the texture object from the current OpenGL rendering context.

3.15.2 Buffer Resource

CBufferGPUResource represents an array of vertices and indices, which are stored in the GPU memory. CBufferGPUResource has two parts. One is for a vertex buffer and the other is for an index buffer.

The vertex buffer consists of one color chunk, one normal chunk, one coordinate chunk and multiple texture coordinate chunks，which store color, normal, coordinate and texture coordinates of the vertices respectively. Struct CGPUChunkInfo stores the information of a chunk.

The index buffer consists of integer numbers. The data type of the integer number is specified by m_IndexComponentType.

The declarations of CBufferGPUResource and related structure are listed as follows:

```
class CBufferGPUResource : public IGPUResource
{
protected:
    GLuint                  m_VertexBufferHandle;
    GLuint                  m_IndexBufferHandle;
public:
    CGPUChunkInfo           m_Color;
    CGPUChunkInfo           m_Normal;
    CGPUChunkInfo           m_Coord;
    TMF<CGPUChunkInfo>      m_TexCoords;

// m_IndexComponentType can be GL_UNSIGNED_BYTE, GL_SHORT or GL_INT.
    GLenum                  m_IndexComponentType;

    virtual void Bind() const
    {
        glBindBuffer(GL_VERTEX_ARRAY,m_VertexBufferHandle);
        if(m_Coord.ChunkPosition>=0)
        {
            glEnableClientState(VERTEX_ARRAY);
            glVertexPointer(m_Coord.ComponentCount,
                    m_Coord.ComponentType, 0, m_Coord.ChunkPosition);
        }
        if(m_Normal.ChunkPosition>=0)
        {
            glEnableClientState(NORMAL_ARRAY);
```

```
            glNormalPointer(m_Normal.ComponentType,0,m_Normal.
            ChunkPosition);
        }
        if(m_Color.ChunkPosition >= 0)
        {
            glEnableClientState(COLOR_ARRAY);
            glColorPointer(m_Color.ComponentCount,
                    m_Color.ComponentType, 0, m_Color.ChunkPosition);
        }
        for(int i=0; i<m_TexCoords.size(); ++i)
        {
            glClientActiveTexture(GL_TEXUTRE0+i);
            glEnableClientState(TEXTURE_COORDINATE_ARRAY);
            glTexCoordPointer(m_TexCoords[i].ComponentCount,
                    m_TexCoords[i].ComponentType,0,
                    m_TexCoords[i].ChunkPosition);

        }

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_IndexBufferHandle);
        glEnableClientState(INDEX_ARRAY);
        glIndexPointer(ComponentType,0,0);

    }

    virtual void Unbind() const
    {
        glBindBuffer(GL_VERTEX_ARRAY,0);
        if(m_Coord.ChunkPosition>=0)
            glDisableClientState(VERTEX_ARRAY);
        if(m_Normal.ChunkPosition>=0)
            glDisableClientState (NORMAL_ARRAY);
        if(m_Color.ChunkPosition>=0)
            glDisableClientState (COLOR_ARRAY);
        for(int i=0; i<m_TexCoords.size(); ++i)
        {
            glClientActiveTexture(GL_TEXUTRE0+i);
            glDisableClientState (TEXTURE_COORDINATE_ARRAY);
        }

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,0);
        glDisableClientState(INDEX_ARRAY);
    }
};
```

```
struct CGPUChunkInfo
{
// The component data type of each element in the chunk
    GLenum  ComponentType;
// The component count of each element in the chunk
    GLint   ComponentCount;
// The start position of the chunk in the buffer.
    GLuint  ChunkPosition;
};
```

3.15.3 Shader Program

So far, we only support shaders written by Cg language. We use CCgProgramGPUObject to encapsulate the handles of profile, programs and parameters, which are created by the Cg runtime environment.

```
class CCgProgramGPUResource : public IGPUResource
{
protected:
    CGprofile       m_Profile;
    CGprogram       m_Program;
    TMF<CGparameter> m_Parameters;
public:
    virtual void Bind(TMF<SFVariable>& ActualParameters) const;
    virtual void Unbind() const;
};
```

The member function Bind requires an array of pointers pointing to actual parameters for the following execution of the shader program. The order of the actual parameter pointers must remain consistent with that of the formal parameters, which are defined in m_Parameters.

3.15.4 GPU Resource Manipulator

CGPUResourceManipulator takes charge of all the IGPUResource life period. The declaration of CGPUResourceManipulator is listed as follows:

```
class CGPUResourceManipulator
{
public:
    CImageTextureGPUResource * GiveResource (CImageTexture*);
```

```
    CCubeMapTextureGPUResource * GiveResource (CComposedCubeMapTexture *);
    CBufferGPUResource * GiveResource (CIndexedTriangleSet*);
    CCgProgramGPUResource * GiveResource (CShaderProgram*);

    IGPUResource * GiveResource(IFeature*);

    SFInt    EstimateMemoryCost(CImageTexture*);
    SFInt    EstimateMemoryCost(CComposedCubeMapTexture*);
    SFInt    EstimateMemoryCost(CIndexedTriangleSet*);
    SFInt    EstimateMemoryCost(CShaderProgram*);

    SFInt    GetMemoryCost(CImageTexture*);
    SFInt    GetMemoryCost(CComposedCubeMapTexture*);
    SFInt    GetMemoryCost(CIndexedTriangleSet*);
    SFInt    GetMemoryCost(CShaderProgram*);

    SFInt    FreeUnusedResource(IFeature*);
//Calculate the total GPU memory allocated for elements in m_Resources.
    SFInt    TotalMemoryCost() const;

    ...
protected:
    Map<IFeature*, IGPUResource*> m_Resources;
}
```

All IGPUResource instances, which have been allocated GPU memory, are placed in the data member Map<IFeature*, IGPUResource * > m_Resources. The keys of this map are the pointers pointing to features that require the conversion from their own data into some GPU resources. The features are also called GPU resource applicants.

For each kind of GPU resource, CGPUResourceManipulator provides a separate version of GiveResource. The input of GiveResource is the GPU resource applicant. It firstly checks the map to see whether it has ever allocated the corresponding GPU resources to this applicant. If so, it retrieves the GPU resource, makes a reference of it, and returns it to the function caller. If not, it allocates GPU memory to create a new IGPUResource instance, inserts it in the map, and returns it.

From the implementation of GiveResource, we know that if the reference count of some GPU resource becomes 0, the resource can be safely recycled.

When a resource applicant does not require the GPU resource any more, FreeUnusedResource (IFeature *) can be called to set the corresponding GPU resources unlinked to the applicant.

The member function EstimateMemoryCost is used to estimate the GPU memory cost before actually allocating the GPU resources required by the applicant. If the required GPU resources have already been allocated to the applicant, it returns 0.

The member function GetMemoryCost counts the GPU memory cost for the GPU resources allocated to the applicant. If the applicant has ever applied for the corresponding GPU resources, it returns 0.

Next, we use an example of CImageTextureGPUResource to explain how to allocate GPU resources for an applicant:

```
CImageTextureGPUResource * CGPUResourceManipulator::GiveResource
    (CImageTexture * p)
{
    if(p == NULL) return NULL;
    if(m_Resources[p])
        return (CImageTextureGPUObject *)m_pResources[p]->Reference();

    CImageTextureGPUResource * pGR;
    Create the OpenGL texture object for *p and let TexID be the identifier.
    if(TexID != 0)
    {
        Create a new object *pTexGR of CImageTextureGPUResource;
        pTexGR->m_Handle = TexID.
        pTexGR->m_GPUMemoryCost = the memory cost of the OpenGL texture
        object.
        m_Resources.Insert(p, pTexGR);
        return pTexGR;
    }else{
        return NULL;
    }
}
```

IGPUResource * GiveResource(IFeature *) is another version of GiveResource. It will check the concrete subclass type of IFeature at first, and then do the allocation by the subclass type.

The pseudo codes of member function FreeUnusedResource are listed as follows.

```
SFInt CGPUResourceManipulator::FreeUnusedResource (IFeature * pFeature)
{
    if(pFeature == NULL)
        return 0;
    pGPUResource = m_Resources[pFeature];
    if(pGPUResource == NULL)
        return 0;

    pGPUResource->Unreference();
    if(pGPUResource ->GetReferenceCount() <= 0)
    {
        GPUMemoryCost = pGPUResource->GetGPUMemoryCost();
        Free GPU memory occupied by *pGPUResource;
        Delete the object *pGPUResource;
        m_Resources.Remove(pFeature);
        return GPUMemoryCost;
    }else{
        return 0;
    }
}
```

After the GPU resource corresponding to the applicant pointed by pFeature is found, Unreference is called once. If the reference count becomes zero, the GPU resource is released, the applicant is removed from the map and the size of released GPU memory is returned. If the reference count is larger than zero, we just return 0.

3.16 Render Target and Its Manager

CRenderTarget is a wrap for Frame Buffer Object (FBO) of OpenGL. FBO is a kind of extension of OpenGL. A more detailed introduction to FBO can be found on the home page at http://www.opengl.org/registry/. Currently, most graphics cards support FBO.

```
class CRenderTarget
{
protected:
    SFUInt32         m_FBOId;

    SFInt32          m_OutputColorTextureCount;
    SFUInt32         m_DepthTextureId;

    TMF<SFUInt32>    m_OutputColorTextureIds;
    SFBool           m_bOutputDepthTexture;

    SFUInt32         m_Width, m_Height;
    SFBool           m_bAntiAlias;
    CWinInfo *       m_WinInfo;
public:
    voidMakeCurrent();

    int SetOutputDepthTexture(SFBool);
    int SetOutputColorTextureCount(SFInt32 Count);

    int CreateRenderTexture();
    int CreateRenderWindow(const CWinInfo& WinInfo);
    …
}
```

The data member m_FBOId records the identifier of FBO. If m_FBOId is equal to 0, it means that the render target is a render window. That means the contents in the frame buffer will be output to the window. If m_FBOId is larger than 0, the render target is a render texture in GPU memory. That means the contents in the frame buffer will be output to a texture.

The data member m_OutputColorTextureIds records an array of texture identifiers. It is used to cope with the case where multiple color buffers exist in a

single frame buffer (OpenGL 2.0 allows this case). Each texture identifier represents a texture that is used to store one color buffer in the frame buffer. The data member m_DepthTextureId records a texture identifier that represents a depth buffer texture of the frame buffer.

The data member m_OutputColorTextureCount specifies how many color textures are required by outputting the color buffers from the frame buffer. The data member m_bOutputDepthTexture indicates whether the depth texture is required by outputting the depth buffer from the frame buffer.

The data member m_Width and m_Height specify the size of the render target. The data member m_bAnitAlias indicates whether anti-alias is required.

The member function MakeCurrent is used to make this render target be the active rendering context and be the destination of the following rendering outputs. The codes are listed as follows:

```
voidCRenderTarget::MakeCurrent()
{
    if(m_pWinInfo && m_pWinInfo->m_hRC)
        glMakeCurrent(::GetDC(m_pWinInfo->m_hWnd), m_pWinInfo->m_hRC);
    else if(m_FBOId > 0)
        glBindFramebufferEXT(m_FBOId);
}
```

Aiming at two kinds of render target, render window and render texture, CRenderTarget provides two methods to create render window and texture.

The implementation of CreateRenderWindow is simple. Only m_FBOId is set to be equal to 0 and an appropriate setting according to the window information is made.

CreateRenderTexture does rather complicated work. It first generates a frame buffer object, then applies for some blank textures in GPU memory, and next binds these textures to the frame buffer object so that the buffer contents of the frame buffer object can be output to the corresponding textures respectively.

Before calling CreateRenderTexture, developers can specify whether the contents of the depth buffer are required and the contents of how many color buffers are required, by using SetOutputDepthTexture and SetOutputColorTextureCount. By default, only one color buffer is output.

The pseudo codes of CreateRenderTexture are listed as follows:

```
int CRenderTarget::CreateRenderTexture()
{
    If current graphics hardware does not support FBO extension
        return 0;

    Create a Frame Buffer Object, whose ID is m_FBOId;
    glGenTextures(m_OutputColorTextureCount,
    m_OutputColorTextureIds.GetBuffer());
    TMF<GLenum> ColorBuffers;
```

```
    for(i=0; i<m_OutputColorTextureIds.size();++i)
    {
        TextureId = m_OutputColorTextureIds[i];
        // Specify the texture that would be attached to the frame buffer
        // object to store one of color buffers.
        glBindTexture(TextureId);
        glTextureImage2D(GL_TEXTURE_2D,...,m_Width,m_Height,...,NULL);
        glBindFramebufferEXT(m_FBOId);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                  GL_COLOR_ATTACHMENT0_EXT + i,
                                  GL_TEXTURE_2D,
                                  TextureId,0);
        ColorBuffers.push_back(GL_COLOR_ATTACHMENT0_EXT + i);
    }

    if(m_bOutputDepthTexture)
    {
        glGenRenderbuffersEXT(1, &m_OutputDepthTextureId);
        glBindRenderbufferEXT(m_OutputDepthTextureId);
        glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT,
                                 GL_DEPTH_COMPONENT,
                                 m_Width, m_Height);
        glBindFramebufferEXT(m_FBOId);
        glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
                                 GL_DEPTH_ATTACHMENT_EXT,
                                 GL_RENDERBUFFER_EXT,
                                 m_OutputDepthTextureId);
    }
    glDrawBuffers(ColorBuffers.size(), ColorBuffers.GetBuffer());

    If the states of OpenGL are OK
        return 1;
    else
        return 0;
}
```

Inside CRenderTargetManager, there is a map which is used to manage render targets by names. It is similar to the map inside CCameraManager.

```
class CRenderTargetManager
{
public:
    CRenderTarget * Give(const SFString& CameraName);
    CRenderTarget * operator [](const SFString& CameraName);
    ...
protected:
    Map<SFString, CRenderTarget> m_RenderTargets;
};
```

3.17 Render Control Unit

The base class for both pre-renders and render pipelines is IRenderControlUnit. By connecting IRenderControlUnit instances in different ways, we configure various control flows inside a rendering engine. The object represented by IRenderControlUnit instance is called Render Control Unit, or RCU for short.

```
class IRenderControlUnit
{
friend class CRenderingEngine;
protected:
// called by CRenderingEngine
     virtual void _DoPerFrame(CFrameInfo * pFI);
     virtual void _DoBeforeFirstFrame(CFrameInfo * pFI);
     virtual void _DoAfterLastFrame(CFrameInfo * pFI);
public:
// Event handlers
     virtual void PerFrame(CFrameInfo * pFI);
     virtual void BeforeFirstFrame(CFrameInfo * pFI);
     virtual void AfterLastFrame(CFrameInfo * pFI);
     virtual void PostChildrenPerFrame(CFrameInfo * pFI);
     virtual void PostChildrenBeforeFirstFrame(CFrameInfo * pFI);
     virtual void PostChildrenAfterLastFrame(CFrameInfo * pFI);

// Overridables
    virtual IResult * GetResult(){return NULL;}

     virtual void Stop (){m_bStopped = true;}
     virtual void Start(){m_bStopped = false;}
     bool IsStopped() const{return m_bStopped;}

     void    SetNextRCU(IRenderControlUnit* pNextRCU);
     void AppendChildrenRCU(* pRCUnit);

protected:
     bool m_bStopped;
     IRenderControlUnit* m_pNextRCU;
     TMF<IRenderControlUnit*> m_ChildrenRCU;
};
```

Inside IRenderControlUnit, there are three protected virtual functions,
_DoPerFrame,
_DoBeforeFirstFrame,
_DoAfterLastFrame.
At runtime, they are invoked by the rendering engine (or say CRenderingEngine

instance) three times. _DoPerFrame is invoked before each frame starts; _DoBeforeFirstFrame is invoked before the first frame for scene model starts; _DoAfterLastFrame is invoked after the last frame for scene model ends.

The most important control flow inside a rendering engine is the sequence for executing_DoPerFrame of different render control units. For simplicity, our framework has the same sequences for executing _DoBeforeFirstFrame, _DoAfterLastFrame and _DoPerFrame.

In IRenderControlUnit, the data member IRenderControlUnit* m_pNextRCU points to another RCU which is connected to the rear of this RCU in series. Namely, m_pNextRCU->_DoPerFrame will be invoked just after this->_DoPerFrame is invoked.

The data member m_ChildrenRCU is an array of pointers, each of which points to a child RCU of this RCU. In other words, during the execution of this->_DoPerFrame, each child RCU's _DoPerFrame will be called one by one.

Through the pseudo codes of _DoPerFrame, readers can see clearly the control flow represented by render control units. _DoPerFrame invokes two event handlers, PerFrame and PostChildrenPerFrame. PostChildrenPerFrame is invoked after all child RCUs' _DoPerFrame are executed.

```
struct CFrameInfo
{
    SFInt64 index; // the frame index
    SFInt64 ms; // the frame beginning time in milliseconds
};

void IRenderControlUnit::_DoPerFrame(CFrameInfo * pFI)
{
    if(IsStopped()) return;
    PerFrame(pFI);
    for(i=0; i< m_ChildrenRCU.size();++i)
        m_ChildrenRCU[i]->_DoPerFrame (pFI);
    if(m_ChildrenRCU.size() > 0)
        PostChildrenPerFrame(pFI);

    if( m_pNextRCU)
        m_pNextRCU->_DoPerFrame(pFI);
}
```

Fig. 3.31 shows an example of the connection of 5 RCUs. In this example, RCU_1.m_pNextRCU points to an RCU_2, RCU_1.m_ChildrenRCU contains two children RCU_{11} and RCU_{12}. RCU_2 contains one child RCU_{21}. The solid lines with arraw represent the control flow, or let us say the execution sequence of _DoPerFrame invoked by RCU_1._DoPerFrame.

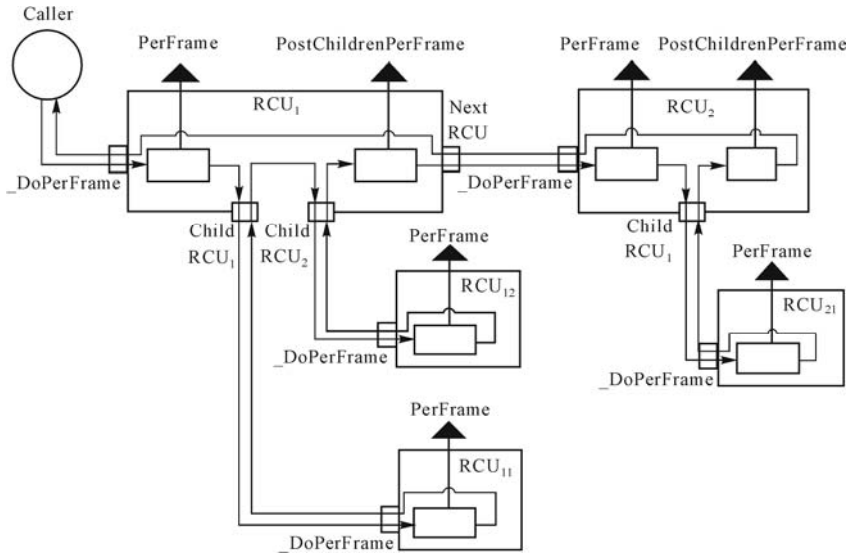

Fig. 3.31 The control flow among 5 render control units

The pseudo codes of functions _DoBeforeFirstFrame and _DoAfterLastFrame are similar to that of the function _DoPerFrame. It only invokes different event handlers.

```
void IRenderControlUnit::_DoBeforeFirstFrame (CFrameInfo*pFI)
{
    if(IsStopped()) return;
    BeforeFirstFrame (pFI);
    for(i=0; i< m_ChildrenRCU.size();++i)
        m_ChildrenRCU[i]->_DoBeforeFirstFrame(pFI);
    PostChildrenBeforeFirstFrame (pFI);
    if( m_pNextRCU)
        m_pNextRCU->_DoBeforeFirstFrame (pFI);
}
```

The member function GetResult returns the computation results if they exist. IRenderControlUnit only provides a dummy implementation of GetResult, and its meaningful implementation should be provided by the concrete subclasses of IRenderControlUnit.

The member functions Start and Stop are used to control the running state of RCU.

The member function AppendChildrenRCU is used to append a child RCU in the rear of m_ChildrenRCU.

The member function SetNextRCU is used to set the data member m_pNextRCU.

- *Event Handlers*

PerFrame. It is invoked when every frame just starts.

BeforeFirstFrame. It is invoked before the first frame when a scene model starts. It can be used to do some initialization for a newly opened scene model.

AfterLastFrame. It is invoked after the last frame when a scene model ends. It is often used to release rendering resources for the scene model.

PostChildrenPerFrame. For each frame, it is called after executing the current RCU's and all child RCUs' PerFrame, but before executing the next RCU's PerFrame.

PostChildrenBeforeFirstFrame. It is called after executing the current RCU's and all child RCU's BeforeFirstFrame, but before executing the next RCU's BeforeFirstFrame.

PostChildrenAfterLastFrame. It is called after executing the current RCU's and all child RCU's AfterLastFrame, but before executing the next RCU's AfterLastFrame.

3.18 Pre-render and Its Manager

A CPreRenderManager manages one or multiple pre-renders. Every pre-render has the same interface of IPreRender. Pre-renders provide pre-rendering, which is,

(1) Modifying scene smodel, i.e., updating scene graph/spatial index nodes or features, adding new attributes to some nodes or features.

(2) Generating new render queues, in which there are to-be-rendered entities.

(3) Performing a certain computation, and the results can be accessed by other parts inside the render engine.

3.18.1 IPreRender

Each kind of pre-render inherits the interface class IPreRender, which is a subclass of IRenderControlUnit. Fig. 3.32 illustrates the interface IPreRender. In this diagram, we visualize a pre-render as an intergrated circuit chip. It has several data input/output ports. The data input ports include rendering traverse root, camera, render target and multiple render queues. The data output ports include multiple render queues and a pointer to class IResult instance, which can be obtained by calling IRenderControlUnit::GetResult(). On the left side of the chip there are six events, which are the same as those in the base class IRenderControlUnit.

The render context consists of the pointers to the camera manager, render queue manager, pre-render manager, render target manager and render pipeline manager, which are the compositions of the rendering engine.

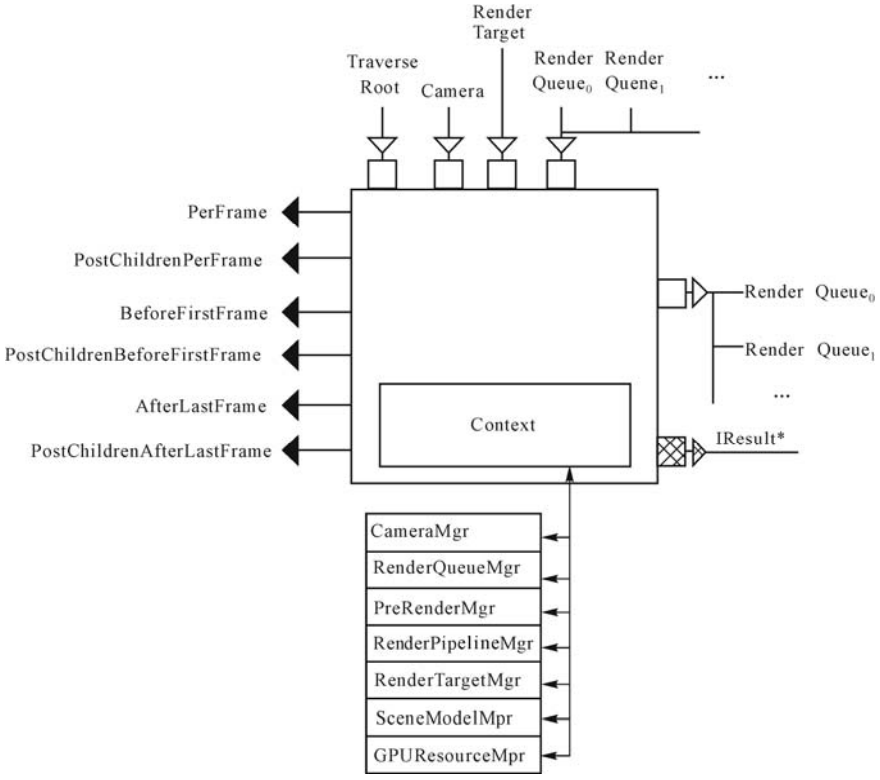

Fig. 3.32 The interface of IPreRender

The declaration and code fragments of class IPreRender are shown as follows:

```
class IPreRender:public IRenderControlUnit
{
public:
// Overridables
    virtual IPreRender * Clone() const = 0;

    // Event handlers that are defined in IRenderControlUnit
    //  virtual void PerFrame(CFrameInfo * pFI);
    //  virtual void BeforeFirstFrame(CFrameInfo * pFI);
    //  virtual void AfterLastFrame(CFrameInfo * pFI);
    //  virtual void PostChildrenPerFrame(CFrameInfo * pFI);
    //  virtual void PostChildrenBeforeFirstFrame(CFrameInfo * pFI);
    //  virtual void PostChildrenAfterLastFrame(CFrameInfo * pFI);
public:
// Set data I/O ports
    void SetInTraverseRoot(IEntity * pRoot);
```

```
    void SetInCamera(CCamera* pCamera);
    void SetInRenderTarget(CRenderTarget * pRT);
    void SetInRenderQueue(int i, CRenderQueue * pRenderQueue);
    void SetOutRenderQueue (CRenderQueue * pRenderQueue);

// Set context
    void SetContext(CRenderingEngine * pRE);

protected:
// The pre-render context, which are set by SetContext
    CSceneModelManipulator2 * m_pSceneModelMpr;
    CGPUResourceManipulator   m_GPUResourceMpr;
    CRenderQueueManager *     m_pRenderQueueMgr;
    CPreRenderManager *       m_pPreRenderMgr;
    CCameraManager *          m_pCameraMgr;
    CRenderTargetManager *    m_pRenderTargetMgr;

// I/O ports
    IEntity *                 m_pInTraverseRoot;
    CCamera *                 m_pInCamera;
    CRenderTarget *           m_pInRenderTarget;
    TMF<CRenderQueue*>        m_InRenderQueues;
    TMF<CRenderQueue*>        m_OutRenderQueues;
};
```

Because class IPreRender inherits from class IRenderControlUnit, IPreRender objects can be easily assembled to form various control flows. In a typical control flow configuration of a rendering engine, there are one or two pre-renders connected in serial and one render pipeline connected to the last pre-render.

3.18.1.1 Data Members

The pointers m_pSceneModelMpr, m_pRenderQueueMgr, m_pPreRenderMgr and m_pCameraMgr, m_pRenderTargetMgr as the pre-render context, point to the corresponding objects in the CRenderingEngine. They are set by the function SetContext.

The pointer m_pInCamera points to a camera object which provides the viewing parameter in the pre-rendering. m_pRenderTarget points to an input render target object.

Since most pre-renderings need a scene traverse, the property m_pInTraverseRoot specifies the root of scene traverses.

Both the array m_InputRenderQueues and m_OutputRenderQueues collect several rendering queue pointers. m_InRenderQueues are inputs, but m_OutRenderQueues are outputs. The function SetInRenderQueue(int, CRenderQueue*) and SetOutRenderQueue

(int, CRenderQueue*) are used to set/add input and output the render queue respectively. The first parameter is the index of the place for the render queue in m_InRenderQueues or m_OutRenderQueues. The array is resized to make sure the index is less than the current size of the array.

3.18.1.2 Clone

The abstract function Clone is defined to support the design pattern, prototype, for extensible instance creation. It will be used by the function CPreRenderManager:: Give.

The implementation of Clone in any concrete subclass of IPreRender creates a new instance, and copies the contents of itself to the new instance.

The following codes show the implementation of Clone:

```
IPreRender CMyPreRender::Clone() const
{
    IPreRender * pNew = new CMyPreRender(*this);
    return *pNew;
}
```

3.18.1.3 Event Handlers

The same as that of IRenderControlUnit.

3.18.2 CPreRenderManager

The main function of CPreRenderManager is to manage pre-render's life periods as well as to provide searching services.

There are two maps, m_Prototypes and m_PreRenders, in CPreRenderManager. The map m_Prototypes stores a variety of pre-render prototypes, and the key is the class name of the prototype. The map m_PreRenders stores the pre-render instances which are referenced in the rendering engine.

A prototype can be registered into m_Prototypes through the function Register. Each prototype can create a new instance by cloning itself, namely by using the function Clone. The design pattern prototype is adopted here. The function Register first makes a copy of the input instance and puts the copy to m_Prototypes. Thus, callers of the function Register should manage the life period of the input instance. The safe method is to use a local variable as the input of the function Register.

The function Give is to yield a named pre-render instance according to a name and a prototype name. If there has been a pre-render with the same name in the map, the function Give directly returns it. Otherwise, the function finds the

prototype according to the specified prototype name and uses the prototype to create a new instance.

The declaration and some code fragments of CPreRenderManager are shown as follows:

```
class CPreRenderManager
{
public:
    void SetRenderingEngine(CRenderEngine * pRE){m_pRenderingEngine =
        pRE;}
    IPreRender * Give(const SFString& PreRenderMame, const SFString&
        PrototypeName);
    voidRegister(const SFString& PrototypeName, const IPreRender*
        pPrototype);
    IPreRender * operator [] (const SFString& PreRenderName);

    void ClearPrototypes(); // Release prototypes in m_Prototypes
    void ClearPreRenders(); // Release pre renders in m_PreRenders
    ...
protected:
    CRenderingEngine * m_pRenderingEngine;
    map< const SFString, IPreRender *> m_Prototypes;
    map< const SFString, IPreRender *> m_PreRenders;
}

IPreRender * CPreRenderManager::Give(const SFString& PreRenderName,
const SFString& PrototypeName)
{
    if m_PreRenders[PreRenderName] exists, then return it;

    if m_Prototypes[PrototypeName] exists
    {
        IPreRender * pPreRender = pProto->Clone();
        m_PreRenders[PreRenderMame] = pPreRender;
        pPreRender->SetContext(this);
        return pPreRender;
    }else
        return NULL;
}

IPreRender*CPreRenderManager::operator[](const SFString& PreRenderName)
{
    if m_PreRenders[PreRenderName] exists, then return it;
    else return NULL;
}
```

3.19 Render Pipelines and Its Manager

In the rendering engine, pre-renders output a set of render queues. To render the elements of render queues into render targets is the main function of rendering pipelines.

3.19.1 IRenderPipeLine

Fig. 3.33 visualizes the interface of IRenderPipeline. Readers can see that the interface is similar to that of IPreRender. The main differences between IRenderPipeline interface and IPreRender interface are listed as follows:

(1) IRenderPipeline has two kinds of outputs, render target and render queues. However, IPreRender has only one kind of output, render queues, since the main function of a render pipeline is to draw objects to a render target.

(2) IRenderPipeline has no need for the input Traverse Root, since a render pipeline only needs to traverse render queues instead of the scene model.

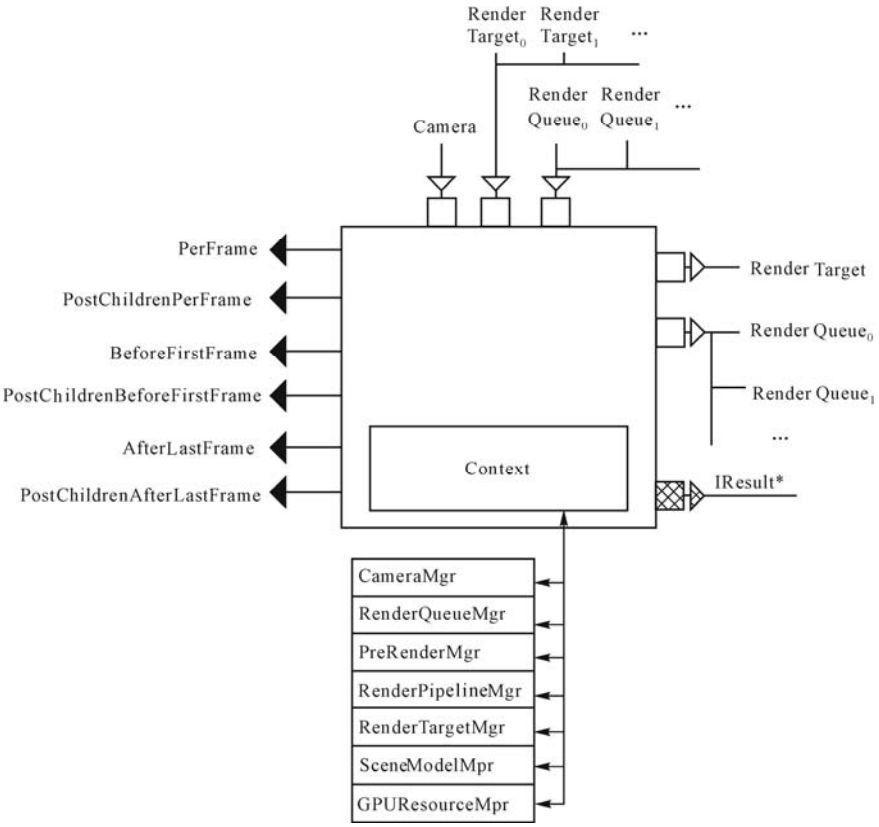

Fig. 3.33 The interface of IRenderPipeline

The declaration of IRenderPipeline is listed as follows:

```
class IRenderPipeline: public IRenderControlUnit
{
public:
// Overridables
    virtual IRenderPipeline * Clone() const = 0;

// Event handlers that are defined in IRenderControlUnit
    //  virtual void PerFrame(CFrameInfo * pFI);
    //  virtual void BeforeFirstFrame(CFrameInfo * pFI);
    //  virtual void AfterLastFrame(CFrameInfo * pFI);
    //  virtual void PostChildrenPerFrame(CFrameInfo * pFI);
    //  virtual void PostChildrenBeforeFirstFrame(CFrameInfo * pFI);
    //  virtual void PostChildrenAfterLastFrame(CFrameInfo * pFI);

// Providing the implementation of PerFrame, which provides the
// framework for drawing all input render queues.
// The main feature of the framework is that it has the rendering budget
// control.
    virtual void PerFrame(CFrameInfo* pFI);

// DrawRenderQueue provides a framework for drawing a render queue,
// and is called during executing PerFrame.
// It traverses the render queue from front to back. For each render
// queue element, it will invoke PrepareElement, HasRemainingBudget
// and DrawElement in order.
    virtual void DrawRenderQueue(CFrameInfo* pFI, SFInt QueueIndex);

// FreeResources is to free the unused resources that have been allocated
// for rendering.
// It is called before DrawRenderQueue during executing PerFrame.
// IRenderPipeline only provides a dummy implementation.
    virtual void FreeResources(CFrameInfo* pFI);

// PrepareElement is to make a render queue element be ready for rendering,
// and is called during executing DrawRenderQueue. If the preparation
// has failed, it will return false and then the element will not be drawn.
// IRenderPipeline only provides a dummy implementation.
    virtual SFBool PrepareElement(CFrameInfo*, CRenderQueueElement*);

// HasBudget checks whether the rendering budget is enough for the
// element rendering and is called after PrepareElement during executing
// PerFrame. If the return value is true, DrawElement is going to be
// called. Otherwise it is not.
// IRenderPipeline only provides a dummy implementation.
```

```
    virtual SFBool HasRemainingBudget (CFrameInfo*, CRenderQueueElement*);

// DrawElement is to draw a render queue element, and is called after
// HasRemainingBudget during executing DrawRenderQueue.
// Whether it is called or not relies on the return value of HasRemainingBudget.
// IRenderPipeline only provides a dummy implementation.
    virtual void DrawElement(CFrameInfo*, CRenderQueueElement*);

// PostDrawElement is invoked after calling DrawElement.
// The last parameter RenderCost is to tell the render cost of the
// recent call of DrawElement.
    virtual void PostDrawElement(CFrameInfo*, CRenderQueueElement*,
    CRenderCost& RenderCost);

public:
// Set data I/O ports
    void SetInCamera(CCamera* pCamera);
    void SetInRenderTarget(int i, CRenderTarget * pRT);
    void SetInRenderQueue(int i, CRenderQueue * pRenderQueue);
    void SetOutRenderTarget (CRenderTarget * pRenderTarget);
    void SetOutRenderQueue(int i, CRenderQueue* pRenderQueue);


// Set context
    void SetContext(CRenderingEngine * pRE);

// Set budget for rendering
    voidSetBudget(const CRenderCost&);
protected:
// The pre-render context, which is set by SetContext
    CSceneModelManipulator2 * m_pSceneModelMpr;
    CGPUResourceManipulator   m_GPUResourceMpr;
    CRenderQueueManager *     m_pRenderQueueMgr;
    CPreRenderManager *       m_pPreRenderMgr;
    CRenderPipelineManager*   m_pRenderPipelineMgr;
    CCameraManager *          m_pCameraMgr;
    CRenderTargetManager *    m_pRenderTargetMgr;

// I/O ports
    CCamera *                 m_pInCamera;
    CRenderTarget *           m_pInRenderTarget;
    TMF<CRenderQueue*>        m_InRenderQueues;
    CRenderTarget *           m_pOutRenderTarget;
    TMF<CRenderQueue*>        m_OutRenderQueues;

// Budget for rendering
    CRenderCost               m_Budget;
    CRenderCost               m_RemainingBudget;
```

```
// The cost for recent draw
    CRenderCost               m_RecentDrawCost;
};

struct CRenderCost
{
    SFTime           m_Time;
    SFInt            m_GPUMemory;
};
```

The declarations of IRenderPipeline and IPreRender are similar. Readers can refer to the introduction of IPreRender.

IRenderPipeline is also a subclass of IRenderControlUnit. It means IRenderPipeline instances can be connected both in series and in hierarchy to form a parent-children relationship.

In ordinary cases, several IRenderPipeline instances in series connection follow several IPreRender instances in series connection.

- *Default Implementation of PerFrame*

The event handler PerFrame is invoked per frame. The concrete subclasses of IRenderPipeline should provide a meaningful implementation of this event handler to realize the drawing of all render queue elements to the render target. However, to reduce the efforts to develop a concrete render pipeline, IRenderPipeline provides a default implementation of the event handler PerFrame.

In the default implementation, the first step is to prepare the output render target. Then, the viewing parameters of the final output are set according to the camera pointed by m_pInCamera. The next step is to release unused rendering resources through calling the overridable function FreeResources. Afterwards, the rendering budget is updated. At last, for each render queue, the function DrawRenderQueue is called once.

The pseudo codes are listed as follows:

```
void IRenderPipeline::PerFrame(CFrameInfo* pFI)
{
    Prepare the render target *m_pOutRenderTarget.
    Set viewing parameters according to *m_pInCamera;

    FreeResources(pFI);
// Update m_RemainBudget
    m_RemainBudget.m_Time = m_Budget.m_Time – the time elapse since
    the frame begins;
    m_RemainBudget.m_GPUMemory = m_Budget.m_GPUMemory —
                                 m_GPUResourceMpr.TotalMemoryCost().

    for(i=0; i< m_InRenderQueues.size(); ++i)
    {
```

```
            DrawRenderQueue (pFI,i);
        }
    }
```

The implementation of IRenderPipeline::FreeResources does nothing. The concrete subclasses should provide meaningful implementation. Placing FreeResources ahead of actual drawing will benefit the following drawing procedures by providing as many rendering resources as possible. Additionally, since all to-be-rendered objects are known (all in render queues), it is possible for this function to check the usage of rendering resources to assure that only unused rendering resources are freed.

The overridable member function DrawRenderQueue is to draw the elements in a render queue. For each element, DrawRenderQueue first calls the overridable function PrepareElement to make a certain preparation, then calls the overridable function HasRemainingBudget to estimate whether the remaining rendering budget for time and memory space is enough to render the current element. If the remaining render budget is enough, DrawRenderQueue calls the overridable function DrawElement to perform drawing. DrawRenderQueue measures the execution time and increased GPU memory consumption for DrawElement, and uses the measurement results to update the rendering budget. Eventually, the measurement results as the rendering cost are passed into the overridable function PostDrawElement.

The pseudo codes of DrawRenderQueue are listed as follows:

```
void IRenderPipeline::DrawRenderQueue (CFrameInfo* pFI, SFInt QueueIndex)
{
    SFInt IncreasedGPUMemorySize;
    For each element E in *m_InRenderQueue[QueueIndex]
    {
        if(PrepareElement(pFI,&E) && HasRemainingBudget(&E))
        {
            SFInt   GPUMem1 = m_GPUResourceMpr.TotalMemoryCost();
            SFTime  Time1 = GetCurrentTime();
            DrawElement(pFI, &E);

            SFInt GPUMem2 = m_GPUResourceMpr.TotalMemoryCost();
            SFTime  Time2 = GetCurrentTime();

            m_RecentDrawCost.m_Time = Time2 – Time1;
            m_RecentDrawCost.m_GPUMemory = GPUMem2 – GPUMem1;
            m_RemainingBudget –= m_RecentDrawCost;

            PostDrawElement(pFI, &E, &m_RecentDrawCost);
        }
    }
}
```

It is apparent that IRenderPipeline provides a rendering framework which can control rendering time and memory consumption. The meaningful codes of the framework are in the default implementation of PerFrame and DrawRenderQueue. Ordinarily, it is not necessary to override PerFrame and DrawRenderQueue. The following functions should be overridden for the concrete render pipelines, which are

PrepareElement,
HasRemainingBudget,
DrawElement,
PostDrawElement,
FreeResources.

3.19.2 Modular Render Pipeline

In this subsection we will introduce a subclass of render pipeline, called modular render pipeline. As the name indicates, a modular render pipeline consists of a group of render modules. We are going to use a metaphor to explain it. A modular render pipeline could be regarded as a punching machine, which is able to punch many kinds of artifacts. A render queue is a conveyor belt, and a render queue element is a raw material blank labelled with a punching specification. Different features corresponding to a render queue element are represented by blanks with different materials. A render module is a punching mold, which can be installed in the punching machine. For punching different artifacts, different punching molds should be used. Then, the rendering procedure of a render pipeline is like producing artifacts by the punching machine. For each input blank, the machine can select an appropriate punching mold according to the blank's material types and punching specifications. This selection procedure is programmable. Moreover, if a punching machine has installed more types of punching molds, it can handle more types of blanks, and produce more types of artifacts. It is the same for a modular render pipeline. A modular render pipeline armed with more kinds of render modules can process more types of features and synthesize more rendering effects.

IModularRenderPipeline is the interface class of various modular render pipelines. The declaration is listed as follows:

```
class IModularRenderPipeline : public IRenderPipeline
{
public:
// Overridable functions
    virtual IRenderPipeline * Clone() const = 0;
    virtual IRenderModule * MatchRenderModule(IFeature*) = 0;
public:
```

```
    void SetRenderModuleAttributePosition(SFInt AttributePosition);
public:
// Provide meanful implementations for the overridable functions
    virtual SFBool PrepareElement(CFrameInfo*, CRenderQueueElement*);
    virtual SFBool HasRemainingBudget (CFrameInfo*,CRenderQueueElement*);
    virtual void DrawElement(CFrameInfo*, CRenderQueueElement*);
    virtual void FreeResources(CFrameInfo *);

    IRenderModuleAttribute* GetRenderModuleAttribute
        (CRenderQueueElement*);
    voidSetRenderModuleAttribute(CRenderQueueElement*,
        IRenderModuleAttribute*);
    …
protected:
    IRenderModule *           m_pCurrentRenderModule;
    TMF<IFeature*>            m_CurrentGPUResourceApplicants;
    SFInt                     m_RenderModuleAttributePosition;

// m_RenderedFeatureCache stores some rendered features using the LRU
// strategy.
    TMF<IFeature*>            m_RenderedFeatureCache;
};
```

IModularRenderPipeline overrides the following five functions to perform relevant tasks:

(1) PrepareElement. It selects an appropriate render module for the to-be-rendered render queue element.

(2) HasRemainingBudget. It uses the selected render module to estimate the rendering cost, and checks if there is enough remaining rendering budget to cover the cost.

(3) DrawElement. It applies for relevant rendering resources and draws the element by using the selected render module. Meanwhile, the owner of the rendering resources is kept in m_RenderedFeatureCache.

(4) PostDrawElement. It updates the rendering cost in the attribute of the rendered element.

(5) FreeResources. According to m_RenderedFeatureCache, the least recently unused rendering resources are released.

The remaining part of this subsection introduces these functions in detail.

3.19.2.1 PrepareElement

The pseudo codes of PrepareElement are as follows:

```
SFBool IModularRenderPipeline::PrepareElement(CFrameInfo*,
    CRenderQueueElement* pE)
{
    IRenderModuleAttribute * pAttrib;
    pAttrib = GetRenderModuleAttribute(pE);
    if(pAttrib){
        m_pCurrentRenderModule = pAttrib->m_pRenderModule;
        return NULL != m_pCurrentRenderModule;
    }else{
        m_pCurrentRenderModule = MatchRenderModule(pE->pFeature);
        if(m_pCurrentRenderModule){
            pAttrib = m_pCurrentRenderModule->CreateAttribute(pFeature);
            pAttrib->m_pRenderModule = m_pCurrentRenderModule;
            SetRenderModuleAttribute(pE, pAttrib);
            return true;
        }else{
            return false;
        }
    }
}
```

PrepareElement finds a proper render module to match the feature of the input render queue element and records the pointer in an IRenderModuleAttribute instance, which is further linked to the element through the function SetRenderModuleAttribute.

The class IRenderModuleAttribute is declared as follows:

```
class IRenderModuleAttribute:public class IAttribute
{
public:
    virtual ~IRenderModuleAttribute(){}
private:
    IRenderModule *  m_pRenderModule;
    …
    friend class IModularRenderPipeline;
}
```

Besides a pointer to the render module, IRenderModuleAttribute has other data members, which will be introduced together with their usage in the following subsections.

In most cases, each kind of render module should define its own version of attribute, namely a concrete subclass of IRenderModuleAttribute. Then, the developers for render modules can freely define attributes that are in friendly

forms to these render modules. Since the framework has no idea of any concrete render module attribute, each concrete render module must provide a method to produce new attribute instances, as the function IRenderModule::CreateAttribute we will see later.

The matched render module for a specified feature is pointed to by the pointer IRenderModuleAttribute::m_pRenderModule. Moreover, this attribute instance will be inserted in the attribute array m_AttriSet of the feature. The index of the attribute in the array is indicated by IModularRenderPipeline::m_RenderModuleAttributePosition. Developers can use the following two functions to get and set the attribute respectively.

```
IRenderModuleAttribute*
IModularRenderPipeline::GetRenderModuleAttribute(CRenderQueueElement*pE)
{
    return pE->m_pFeature->GetAttribute(m_RenderModuleAttributePosition);
}

void
IModularRenderPipeline::SetRenderModuleAttribute(CRenderQueueElement*
    pE, IRenderModuleAttribute*pAttrib)
{
    delete GetRenderModuleAttribute(pE);
    pE->m_pFeature->SetAttribute(m_RenderModuleAttributePosition,
        pAttrib);
}
```

Fig. 3.34 illustrates the navigation from a feature to its matched render module.

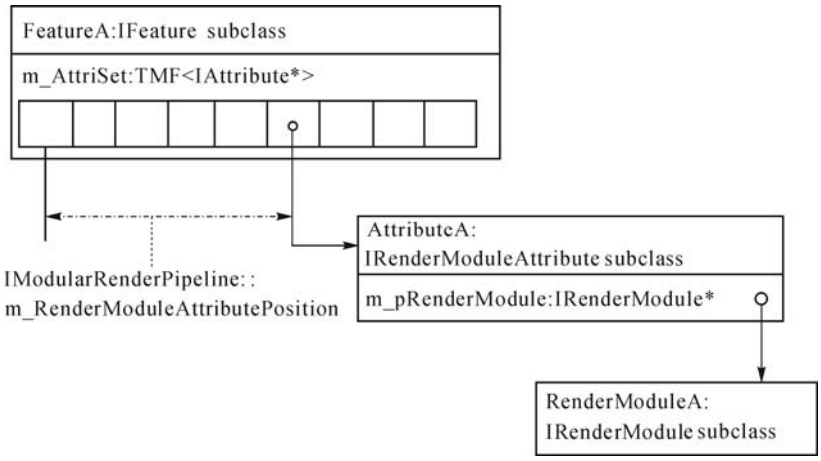

Fig. 3.34 The navigation from a feature to its matched render module

The function PrepareElement first checks if the feature has been matched. If not, the abstract function MatchRenderModule is invoked to find a matched render module. Then, an attribute instance is created by the render module and is used to record the pointer to the matched render module for fast matching the next time.

Obviously, the abstract function MatchRenderModule performs render module matching. Since it is abstract, the concrete subclasses of IModularRenderPipeline should provide the matching strategy in MatchRenderModule's implementation.

3.19.2.2 HasRemainingBudget

The core task of the function HasRemainingBudget is to estimate the time and memory cost for drawing the input element, and make comparison between the cost and the remaining rendering budget. If the budget is not enough, the function returns false value, which means this element is not to be drawn. If the budget is enough, the return value is true and the feature will be drawn.

The pseudo codes of the function HasRemainingBudget are as follows:

```
SFBool IModularRenderPipeline::HasRemainingBudget(CFrameInfo* pFI,
CRenderQueueElement* pE)
{
    CRenderCost Cost;
//Collect the resource application from the corresponding render module.
//Then, estimate GPU memory cost according to the resource application.
    SFInt   GPUMemoryCost =0;
    IRenderModuleAttribute*pAttrib=GetRenderModuleAttribute(pE);
    m_CurrentGPUResourceApplicants.Clear();
    m_pCurrentRenderModule->ApplyForGPUResources(pFI, m_pInCamera,
        pE, pAttrib, m_CurrentGPUResourceApplicants);
    for each pointer to feature pFeature in m_CurrentGPUResourceApplicants
    {
        GPUMemoryCost += m_GPUResourceMpr->EstimateMemoryCost (pFeature);
        if(pFeature does not exist in pAttrib->m_GPUResourceApplicants)
            pAttrib->m_GPUResourceApplicants.Add(pFeature);
    }

    m_pCurrentRenderModule->EstimateCost(m_pInCamera,pE,pAttrib,TimeCost);

    Cost.m_Time = TimeCost;
    Cost.m_GPUMemory = GPUMemoryCost;

    // Check the remaining budget is enough to cover the cost either
    // in memory or time.
    if(m_RemainingBudget >= Cost)
        return true;
    else
        return false;
}
```

The data members in IRenderModuleAttribute and relevant to the function HasRemainingBudget are listed as follows:

```
class IRenderModuleAttribute : public class IAttribute
{
…
private:
// m_GPUResourceApplicants stores all GPU resource applicants
// for the corresponding feature.
    TMF<IFeature*>   m_GPUResourceApplicants;

    friend class IModularRenderPipeline;
}
```

It should be noted that the remaining time and GPU memory recorded in IModularRenderPipeline::RemainingBudget have different measurement scopes. The rendering budget of time is the allowed maximum time duration for any frame, and the rendering budget of GPU memory is the allowed maximum memory size occupied by rendering pipelines for the entire runtime. Thus, the measurement scope of the remaining time is per frame, but that of the remaining GPU memory is for the entire runtime. Therefore, we use different ways to estimate the time cost and the memory cost.

In current implementation, GPU memory consumption should be estimated for each frame. Note that the remaining GPU memory budget has counted the accumulated actual GPU memory consumption of all previous frames. Therefore, when we estimate GPU memory consumption for one frame, only the GPU memory allocated to this frame should be counted.

The remaining budget m_RemainingBudget is not really decreased after executing the function HasRemainingBudget. It is done by the function IRenderPipeline::DrawRenderQueue after executing the function DrawElement. The reason is that the function HasRemainingBudget only obtains the estimated cost. The actual cost should be measured when executing DrawElement.

The purpose of calling the function m_pCurrentRenderModule->ApplyForGPUResources is to query what GPU resources the current matched render module needs. To realize it, the data member m_CurrentGPUResourceApplicants, the type of which is TMF<IFeature*>, is passed to the function m_pCurrentRenderModule-> ApplyForGPUResources. The current render module fills the array with the pointers to the features that need GPU resources. A feature that requires GPU resources is called a GPU resource applicant. Therefore, the nature of the above procedure is that the framework of a modular render pipeline gives the matched render module a chance to provide a list of GPU resource applicants. By this list, the function HasRemainingBudget can estimate the memory cost by querying the GPU resource manipulator m_GPUResourceMpr. It is done through the function m_GPUResourceMpr.EstimateMemoryCost, where the input parameter is the pointer to a GPU resource applicant. The return value is the size of GPU memory

to be allocated just in this frame for the GPU resources required by the applicant. Or, let us say the estimation is the newly increased GPU memory cost. In a case where the GPU resources that an applicant required have already been allocated to the applicant, the function m_GPUResourceMpr.EstimateMemoryCost returns 0. This case is quite common. For example, there are two shape features that share one image texture. During the rendering of the first shape, as an applicant, the image texture has applied for the corresponding GPU resource successfully. Then, to render the second shape as an applicant, the image texture will apply for resources again. In this case, the GPU resource manipulator will not allocate a new resource for it. Consequently, the GPU memory cost of the texture to render the second shape should be estimated as 0.

Since m_GPUResourceMpr->EstimateMemoryCost can assure no repetitive count of GPU memory consumption by itself, the local variable GPUMemoryCost in the function HasRemainingBudget correctly accumulates the estimation of total GPU memory consumption of all to-be-rendered features in this frame.

All GPU resource applicants owned by the feature of a rendering queue element are recorded in IRenderModuleAttribute::m_GPUResourceApplicants, as is done by the following codes in the function HasRemainingBudget:

```
for each pointer to feature pFeature in m_CurrentGPUResourceApplicants
{
    GPUMemoryCost += m_GPUResourceMpr->EstimateMemoryCost
    (pFeature);
    if(pFeature does not exist in pAttrib->m_GPUResourceApplicants)
        pAttrib->m_GPUResourceApplicants.Add(pFeature);
}
```

3.19.2.3 DrawElement

The implementation of the function DrawElement performs the following tasks:

(1) To Allocate GPU resources according to the applicant list provided by the render module.

(2) To pass the allocated GPU resources to the render module as the input parameter when calling the render module's overridable method PrepareResources.

(3) To call the render module's overridable method Draw.

(4) To assign the features of render queue elements to the data member IModularRenderPipeline::m_RenderedFeatureCache, which records all features which have been rendered so far.

The pseudo codes of DrawElement are as follows:

```
void IModularRenderPipeline::DrawElement(CFrameInfo* pFI,
    CRenderQueueElement* pE)
{
```

```
// Allocate GPU resources according to the resource applicants.
// The pointers to the allocated GPU resources will be placed in
// GPUResourcesForThisFrame in the same order of the applicant in
// m_CurrentGPUResourceApplicants.

  TMF<IGPUResource * > GPUResourcesForThisFrame;
     for each feature *pFeature in m_CurrentGPUResourceApplicants
     {
          IGPUResource * pGPUResource = m_GPUResourceMpr.GiveResource
               (pFeature);
          GPUResourcesForThisFrame.Add(pGPUResource);
     }

 if(m_pCurrentRenderModule->PrepareResources(pFI, m_pInCamera, pE,
  pAttrib, &GPUResourcesForThisFrame))
     {
          m_pCurrentRenderModuel->Draw(pFI, m_pInCamera, pE, pAttrib);
          if pE->m_pFeature does not exist in m_RenderedFeatureCache
               m_RenderedFeatureCache.Add(pE->m_pFeature);
     }
}
```

The function DrawElement first uses the GPU resource manipulator m_GPUResourceMpr to allocate the corresponding resources to the applicants in the list m_CurrentGPUResourceApplicants. One applicant obtains one GPU resource. The allocated GPU resources are recorded in the array GPUResourcesForThisFrame. The GPU resources in GPUResourcesForThisFrame are arranged in the same order as the corresponding applicants are arranged in m_CurrentGPUResourceApplicants.

GPUResourcesForThisFrame is passed to the function m_pCurrentRenderModule-> PrepareResources. If PrepareResources returns true, m_pCurrentRenderModule-> Draw will be exceuted. In most cases, the current render module needs to record some references to the GPU resources in the attribute pointed to by pAttrib through PrepareResources. These references can be used by m_pCurrentRenderModule-> Draw. Additionally, in PrepareResouces the render module performs other preparations including creating some objects and aggregating them into the attribute. Here, the attribute is used to connect a feature and its matched render module, and record all necessary data required by the render module to draw the feature. It keeps the states of render modules independent of to-be-rendered features.

The last step of IModularRenderPipeline::DrawElement is to place the feature in m_RenderedFeatureCache. Each element in m_RenderedFeatureCache points to a feature that has ever been drawn. In other words, each pointed feature owns allocated GPU resources. The function IModularRenderPipeline::FreeResources just uses the cache to release the least recently unused GPU resources. See the part “FreeResources” in this subsection.

3.19.2.4 PostDrawElement

```
void IModularRenderPipeline::PostDrawElement(CFrameInfo* pFI,
CRenderQueueElement* pE, CRenderCost * pRecentDrawCost)
{
    IRenderModuleAttribute * pAttrib = GetRenderModuleAttribute(pE);
    pAttrib->m_MemoryCost += pRecentDrawCost->m_Memory;
    pAttrib->m_TimeCost = pRecentDrawCost->m_Time;
}
```

The function PostDrawElement records the input rendering cost in the attribute corresponding to the rendering element of the pipeline, to count overall rendering costs.

The data members of the class IRenderModuleAttribute relevant to the function PostDrawElement are as follows.

```
class IRenderModuleAttribute : public class IAttribute
{
...
private:
// m_ GPUMemoryCost counts the total GPU memory cost by the corresponding
// feature.
    SFInt            m_GPUMemoryCost;
// m_TimeCost records the time for rendering.
    SFTime           m_TimeCost;
...
}
```

3.19.2.5 FreeResources

The function FreeResources adopts the least recently unused strategy to release the allocated GPU resources. As discussed above, the class IModularRenderPipeline uses the data member, TMF<IFeature*> m_RenderedFeatureCache, to record the features that own the allocated GPU resources.

The pseudo codes of FreeResources are as follows:

```
void IModularRenderPipeline::FreeResources(CFrameInfo* pFI)
{
    IRenderModuleAttribute* pAttrib;
    For each element E in m_InRenderQueues
    {
        pAttrib =GetRenderModuleAttribute(&E);
        pAttrib->m_UsedFrameIndex = pFI->index;
    }
    For each element E in m_RenderedFeatureCache
```

```
    {
        pAttrib =GetRenderModuleAttribute(&E);
        pAttrib->m_UnusedFrameCount = pFI->index –
        pAttrib->m_UsedFrameIndex;
    }
    Sort elements in m_RenderedFeatureCache according to their
    m_UnusedFrameCount in decreasing order;
    For each element E in m_RenderedFeatureCache from front to back
    {
        pAttrib =GetRenderModuleAttribute(&E);
        SFInt FreedGPUMemory = 0;
        For each GPU resource applicant *pA
        in pAttrib->m_GPUResourceApplicants
        {
            FreedGPUMemory += m_GPUResourceMpr->FreeUnusedResource(pA);
        }
        m_RenderedFeatureCache.Remove(&E);
        Delete pAttrib;
        SetRenderModuleAttribute(&E,NULL);
        if the total freed memory FreedGPUMemory is large enough
            return;
    }
}

class IRenderModuleAttribute : public class IAttribute
{
…
public:
    SFInt   GetUsedFrameIndex() const;
    SFInt   GetUnusedFrameCount() const;
private:
// m_UnusedFrameCount and m_UsedFrameIndex are used to free resources.
// They are used in IModularRenderPipeline::FreeResources.
    SFInt   m_UnusedFrameCount;
    SFInt   m_UsedFrameIndex;
    …
    friend class IModularRenderPipeline;
}
```

The function FreeResources first traverses all input render queues and records the current frame index into the attributes corresponding to the render queue elements.

Then it traverses m_RenderedFeatureCache. For each feature in the cache, it calculates how long the feature has not been rendered. The duration is equivalent to the difference between the current frame index and the index of the frame since the feature has never been rendered. The duration is called unrendered duration.

Next, the features in m_RenderedFeatureCache are sorted according to their

unrendered duration. The cache is traversed from the feature with the longest unrendered duration to that with the shortest unrendered duration. For each feature, the GPU resource applicants owned by the feature are found, and the GPU resource manipulator is used to release the GPU resources according to the applicants. After releasing the GPU resources, the feature is removed from the cache and its corresponding attribute is destroyed. The function FreeResources repeatedly processes each feature in this way until the total freed GPU memory exceeds the threshold.

It should be noted that we make a bit of an approximation here. We assume that all render queue elements would be rendered or, let us say the relevant GPU resources would be used. However, in a real situation, probably only a part of render queue elements would be rendered due to the rendering budget control. Therefore, the reserved resources in a real situation occupy more space than what we assume. Obviously, the approximation is conservative.

3.19.3 Render Module

3.19.3.1 IRenderModule

Render modules are bricks of a modular render pipeline. The class IRenderModule is the interface of various render modules. The declaration of IRenderModule is as follows:

```
class IRenderModule
{
public:
virtual IRenderModule * Clone() const = 0;

// Create a new instance of the corresponding render module attribute
// according to the input feature. The input feature is supposed to be
// the to-be-rendered feature.
    virtual IRenderModuleAttribute*CreateAttribute(IFeature*pFeature)=0;

// Estimate time and CPU memory cost for rendering the input CRenderQueueElement
// object.
// @param[in, out] pAttrib. It points to an instance of the concrete
// class of IRenderModuleAttribute.
    virtual void    EstimateCost(
    const CCamera*, const CRenderQueueElement *,
    IRenderModuleAttribute * pAttrib,
    SFTime& TimeCost) = 0;
```

```
// Apply for GPU resources for multiple features, named GPU resource
// applicants.
// Since ApplyForGPUResources is called per frame, the implementation
// of this function should do a certain check to avoid applying for
// the same GPU resources multiple times.
// @param[in, out] pAttrib. It points to an instance of the concrete
// class of
// IRenderModuleAttribute. The function may check the attribute to
// specify which GPU resources
// have ever been applied for.
// @param[out] GPUResourceApplicants. It should be filled with GPU
// resource applicants.
    virtual void ApplyForGPUResources
    (const CFrameInfo*, const CCamera *, const CRenderQueueElement *,
    IRenderModuleAttribute * pAttrib,
    TMF<IFeature*>& GPUResourceApplicants) = 0;

// Prepare resources before drawing render queue element.
// Since PrepareResources is called per frame, the implementation of
// this function should do a certain check to avoid doing the same
// preparation multiple times.
// @param[in, out] pAttrib. It points to an instance of the concrete
// class of IRenderModuleAttribute. The function should set up data
// members in the attribute and these data members will be used in Draw.
// @param[in] GPUResources. The allocated GPU resources for the applicants.
// The order of IGPUResource* in GPUResources is the same as that of
// applicants in GPUResourceApplicants, that is one of the parameters
// of ApplyForGPUResources.
    virtual SFBool PrepareResources
    (const CFrameInfo*, const CCamera *, const CRenderQueueElement *,
    IRenderModuleAttribute * pAttrib,
    const TMF<IGPUResource*>& GPUResources) =0;

// Draw the input render queue element with respect to the input camera.
// @param[in, out] pAttrib. It points to the same instance, as the
// parameter of PrepareResources.
    virtual void Draw
    (CFrameInfo*pFI, CCamera * pCamera, CRenderQueueElement * pElement,
    IRenderModuelAttribute * pAttrib) = 0;
}
```

The usage of the abstract virtual member functions has been introduced in the subsection 3.19.2 “Modular Render Pipeline”.

3.19.3.2 IRenderModuleAttribute

The class IRenderModuleAttribute is tightly related to the class IRenderModule. Different fragments of its declaration have been shown. Here we give the complete declaration. The meaning of the data member can be referred to in the subsection 3.19.2 “Modular Render Pipeline”.

```
class IRenderModuleAttribute : public class IAttribute
{
public:
    virtual ~IRenderModuleAttribute(){}
    SFInt       GetUsedFrameIndex() const;
    SFInt       GetUnusedFrameCount() const;
    SFDouble    GetTimeCost() const;
    SFInt       GetCPUMemoryCost() const;
private:
    IRenderModule *  m_pRenderModule;

// m_GPUMemoryCost counts the total resource used by the corresponding
// feature.
    SFInt           m_GPUMemoryCost;

// m_TimeCost records the time for rendering.
    SFTime          m_TimeCost;

// m_GPUResourceApplicants stores all GPU resource applicants
// for the corresponding feature.
    TMF<IFeature*>   m_GPUResourceApplicants;

// m_UnusedFrameCount and m_UsedFrameIndex are used to free resources.
// They are used in IModularRenderPipeline::FreeResources.
    SFInt           m_UnusedFrameCount;
    SFInt           m_UsedFrameIndex;

friend class IModularRenderPipeline;
}
```

In general, one feature has one attribute, an instance of a subclass of IRenderModuleAttribute. Render modules use both features and attributes to perform the rendering tasks.

3.19.3.3 CRenderModuleManager

The class CRenderModuleManager manages the life periods of a set of render modules. It creates a new render module instance by using the design pattern prototype. Its declaration is listed as follows:

```
Class RenderModuleManager
{
Public:
    IRenderModule * Give(const SFString& Name,
                         const SFString& PrototypeName);
    voidRegister(const SFString& PrototypeName, const IRenderModule
                * pPrototype);
    IRenderModule * operator [] (const SFString& Name);

// Release prototypes in m_Prototypes
    void ClearPrototypes();

// Release Render Pipeline in m_RenderModules
    void ClearRenderModules();

protected:
    map< const SFString, IRenderModule *> m_Prototypes;
    map< const SFString, IRenderModule *> m_RenderModules;
}
```

3.19.4 CRenderPipelineManager

Similar to CPreRenderManager, the class CRenderPipelineManager manages the life periods of a set of render pipelines.

There is only one thing that should be pointed out. The class CRenderPipelineManager aggregates one CRenderModuleManager instance to provide the capability to manage render modules.

There are two reasons why we place a render module manager here. One is that we hope any render module instance can be shared among multiple render pipelines. The other is that we want to lower the coupling of render modules with other components in the rendering engine.

The following is its declaration:

```
Class CRenderPipelineManager
{
Public:
    void SetRenderingEngine(CRenderEngine*pRE)
    {m_pRenderingEngine = pRE;}
    IRenderPipeLine * Give(const SFString& Name,
                         const SFString& PrototypeName);
    voidRegister(const SFString& PrototypeName, const IRenderPipeLine
        * pPrototype);
    IRenderPipeLine * operator [] (const SFString& Name);
```

```
// Release prototypes in m_Prototypes
    void ClearPrototypes();

// Release Render Pipeline in m_RenderPipelines
    void ClearRenderPipeline();

    CRenderModuleManager& GetRenderModuleManager();
Protected:
    CRenderingEngine *                      m_pRenderingEngine;
    Map<const SFString, IRenderPipeLine*>   m_Prototypes;
    Map<const SFString, IRenderPipeLine*>   m_RenderPipelines;

    CRenderModuleManager                    m_RenderModuleMgr;
}
```

3.20 Examples of Pre-render

In this section, we are going to introduce several concrete pre-renders.

3.20.1 CVFCullingPreRender

The class CVFCullingPreRender is designed to perform view frustum culling to form a potentially visible set (PVS), and classify features in PVS into three render queues according to their material. The features in the PVS are first classified into two render queues. The first queue collects opaque features, and the second queue collects translucent features. Next, the features in PVS which have reflective materials are selected to form the third render queue. Such kinds of features are called reflective features. The three render queues are called opaque queue, translucent queue and reflective queue. Note, since a reflective feature is either opaque or translucent, each reflective feature in a reflective queue is also in the opaque queue or translucent queue.

The features in the opaque queue are sorted in front-to-back order with respect to the input camera, but the features in the translucent queue are sorted back-to-front.

The most important function of the class CVFCullingPreRender is the event handler PerFrame, the pseudo codes of which are listed in the following.

```
void CVFCullingPreRender::PerFrame(const CFrameInfo*)
{
    CVFCullingIterator iterator(m_pSceneModelMpr);
    iterator.SetInCamera(m_pInCamera);
```

```
    iterator.SetOutRenderQueues(
        0, m_OutRenderQueues[0],   // Opaque queue
        1, m_ OutRenderQueues[1],  // Translucent queue
        2, m_ OutRenderQueues[2]); // Reflective queue
    iterator.Create(m_pInTraverseRoot);
    iterator.First();
    while(!iterator.IsDone())
        iterator.Next();
}
```

From the pseudo codes, we can see the iterator, a CVFCullingIterator instance, plays the key role. After the traverse, the opaque, translucent and reflective queues are returned by the iterator.

The class CVFCullingIterator inherits CKD2TreeF2BIterator, which traverses the KD tree in front-to-back order. The class CKD2TreeF2BIterator has been introduced in subsection 3.11.1 “Traverse via Iterator”.

The declaration and implementation of the class CVFCullingIterator are listed as follows:

```
class CVFCullingIterator: public CKD2TreeF2BIterator
{
protected:
    enum{Q_Opaque = 0,Q_Translucent,Q_Reflective};
    CRenderQueue * m_Q[3];
    CCamera * m_pCamera;
public:
    void SetCamera(CCamera * pCamera);
    void SetOutQueues(CRenderQueue * pOpaque, CRenderQueue * pTranslucent,
        CRenderQueue * pReflective);
    virtual int OnEntry(void *pParam);
    virtual int OnPrepareDescend(void *pParam);
};

int CVFCullingIterator::OnPrepareDescend(void *pParam)
{
    CKD2TreeComposite * pNode = dynamic_cast<
    CKD2TreeComposite*>(m_pCurrentItem);
    if(pNode == NULL) return 0;

    if the bounding box of *pNode is disjoint from the view frustum
    corresponding to the camera *m_pCamera
        return 0;
    else
        return 1;
}

int CVFCullingIterator::OnEntry(void *pParam)
```

```
{
    CSpatialIndexLeaf* pLeaf = dynamic_cast< CSpatialIndexLeaf*>
    (m_pCurrentItem);
    if(pLeaf == NULL)    return 0;
    for each relation instance *pR in pLeaf->m_Relations
    {
        if the bounding box of the feature *(pR->m_pFeature) is disjoint
        from the view frustum
            return 0;

        CRenderQueueElement Element; Element.m_pMatrix = &
        (pR->m_Transform);
        Element.m_pFeature = pR->m_pFeature;
        if the material of *(pR->m_pFeature) has translucent properties
            m_Q[Q_Translucent].PushFront(Element);
        else
            m_Q[Q_Opaque].PushBack(Element);

        if the material of *(pR->m_pFeature) has reflective properties
            m_Q[Q_Reflective].PushBack(Element);
    }
    return 1;
}
```

The data member CVFCullingIterator::m_pCamera points to the current camera and the data member m_Q keeps an array of pointers to the three output rendering queues.

The class CVFCullingIterator overrides two event handlers OnPrepareDescend and OnEntry.

The event handler OnPrepareDescend first checks whether the current middle node is contained in, or intersected with, the view frustum. If it is, it returns 1, which means that the children nodes of this middle node will be visited recursively. Otherwise, it returns 0, which means no children nodes will be visited.

The event handler OnEntry first checks whether the current node is a KD tree leaf node. If it is, all relations (CRelation_SISG instances) collected by this leaf node are traversed. For each relation, the event handler checks whether the bounding box of the feature referenced by the relation is disjoint from the view frustum. If it is not disjoint, the feature is regarded as a potentially visible feature, and is placed in the corresponding render queue according to its material. If it is disjoint, nothing will be done.

3.20.2 CMirrorPreRender

The class CMirrorPreRender is designed to handle reflective features to synthesize

global reflection shading effects. For convenience, we call reflective features mirrors.

The pre-render works in the following way. Given each mirror in the input queue, the mirrored camera of the input camera is created. Then, a new instance of CMirrorPreRender is created and takes the mirrored camera as its input camera. This new pre-render performs the pre-rendering tasks similar to that of CVFCullingPreRender, such as view frustum culling and material-based classification. The next render control unit (RCU) of the new pre-render is a render pipeline which is for synthesizing the image of the mirror's reflection. The image will be used as a texture mapped to the mirror.

The declaration and implementation of CMirrorPreRender are as follows:

```
class CMirrorPreRender: public CVFCullingPreRender
{
public:
    virtual void PerFrame(const CFrameInfo* pFI)

// Bind the texture in render target to the mirror feature.
    void BindRenderTargetToMirror(CRenderTarget*, IFeature*pMirror);
}

void CMirrorPreRender:: PerFrame(const CFrameInfo* pFI)
{
    CVFCullingPreRender:: PerFrame(pFI);

    CMirrorPreRender MirrorPreRender
    CMyRenderPipeline MirrorRenderPipeline;
    for(each mirror *pMirror in m_OutRenderQueues[Q_Reflective])
    {
        Let MirroredCamera be the mirrored camera of
            *m_pCamera about *pMirror.
        Let Q0, Q1, Q2 be the opaque, translucent and reflective
            render queues respectively.

        MirrorPreRender.SetInTraverseRoot(m_pRenderTraverseRoot);
        MirrorPreRender.SetInCamera(&MirroredCamera);
        Set Q0, Q1, Q2 be the output render queues of MirrorPreRender;

        CRenderTarget pMirrorImage = m_pRenderTarget->Give(Mirror's ID);
        MirrorRenderPipeline.SetInCamera(&MirroredCamera);
        Set Q0, Q1, Q2 be the input render queues of MirrorRenderPipeline;
        MirrorRenderPipeline.SetOutRenderTarget(pMirrorImage);

        MirrorPreRender.SetNextRCU(&MirrorRenderPipeline);
        MirrorPreRender._DoPerFrame(pFI);
        BindRenderTargetToMirror(pMirrorImage, pMirror);
    }
}
```

In this example, we show that developers can use local pre-renders, render queues and render pipelines to perform recursive rendering. In this example, only the render target is a global resource which is managed by the rendering engine. Therefore, under this framework it is quite easy for developers to design a more complicated rendering procedure in this way.

In the above pseudocodes, the class CMyRenderPipeline is just a placeholder. In actual implementation, it can be a concrete class of IRenderPipeline.

3.20.3 COoCEntityLoader

As the name indicates, the class COoCEntityLoader is designed to take charge of loading and unloading out-of-core entities and manage a cache of most used out-of-core entities. The out-of-core entity has been introduced in subsection 3.9.6 “Out-of-Core Entity”.

The pre-render, or more intuitively the loader, first selects the render queue elements which refer to the feature composed of several out-of-core entities. This render queue element is called out-of-core element. Then, for each out-of-core element, the loader checks whether all of the composed out-of-core entities have already been loaded in the host memory. If not, the out-of-core entities that have not been loaded yet are added into a queue, which is called a loading queue. Next, the loading queue is passed to a separate thread, called a loading thread. The loading thread keeps picking entities out of the loading queue and loading the corresponding entities by using the scene model manipulator. The thread is wrapped by the class CLoadingThread.

Readers may feel curious as to how we add a not-loaded entity into the loading queue. There is one thing we should explain clearly. We know any out-of-core entity has an in-core part and an out-of-core part, which is discussed in subsection 3.9.6 “Out-of-Core Entity”. The in-core part is always in memory since the scene model is opened. Therefore, when we say an out-of-core entity has not been loaded, the exact meaning is that the out-of-core part of the out-of-core entity has not been loaded. So each element in the loading queue is a pointer to an IOutOfCoreObject instance, which only contains the in-core part.

It is a fact that loading an entity is much slower than rendering it. For each frame the rendering thread cannot wait to thread and finish loading all entities, which are supposed to be drawn in this frame. Therefore, when the loading thread is still working for the i-th frame, the rendering thread has requested the relevant out-of-core entities for the (i+1)-th frame. Moreover, the to-be-loaded entities for the i-th frame are possibly different from those for the (i+1)-th frame. So it is neither possible nor necessary for the loading thread to load all to-be-loaded entities. Therefore, we set a threshold to control the maximum number of actual to-be-loaded entities per frame.

Obviously, merely counting on loading data directly from a slow external memory per frame cannot meet the real-time rendering requirements. The class CLoadingThread maintains a cache to temporarily store the most recently used out-of-core entities. The least recently unused entities will be discarded.

The declaration and implementation of the class COoCEntityLoader are listed as follows:

```
class COoCEntityLoader:public IPreRender
{
public:
    virtual void PerFrame (const CFrameInfo* pFI)
    virtual void BeforeFirstFrame(CFrameInfo * pFI);
    virtual void AfterLastFrame(CFrameInfo * pFI);
    virtual void Stop ();
    virtual void Start();
protected:
    virtual void GenerateLoadingQueue(TMF<Feature*>&);

protected:
    SFUInt                  m_MaxLoadingQueueSize;
    CLoadingThread *        m_pThread;

    class CLoadingThread
    {
        …
    public:
        void    CopyLoadingQueue(const TMF<IOutOfCoreObject*>&,
            const CFrameInfo*);
        SFInt   Service(void);
        …
    protected:
        TMF<IOutOfCoreObject*>   m_LoadingQueue;
        TMF<IOutOfCoreObject*>   m_LoadedEntitiesCache;
    };
}

void COoCEntityLoader:: PerFrame(const CFrameInfo*)
{
    TMF<IFeature*> LoadingQueue;
    GenerateLoadingQueue(LoadingQueue);
    pThread->CopyLoadingQueue(LoadingQueue, pFrameinfo);
}

void COoCEntityLoader::GenerateLoadingQueue(TMF<Feature*>& LoadingQueue)
{
    while LoadingQueue.Size() < m_MaxLoadingQueueSize
```

```
    {
        Find the feature that has out-of-core entities and is more
        important than others in the input rendering queue
        Collect the feature's out-of-core entities
        for each of these out-of-core entities that are not entirely
        loaded in the memory
            Push the out-of-core entity identifiers to the back of the
            LoadingQueue.
    }
}

void COoCEntityLoader:: BeforeFirstFrame (const CFrameInfo*)
{
    m_pThread->StartThread();
}

void COoCEntityLoader:: AfterLastFrame (const CFrameInfo*)
{
    m_pThread->StopThread();
    m_pThread->Clear();
}

void COoCEntityLoader:: Start (const CFrameInfo*)
{
    m_bStopped = false;
    m_pThread-> StartThread();
}

void COoCEntityLoader:: Stop (const CFrameInfo*)
{
    m_bStopped = true;
    m_pThread-> StopThread ();
}
```

The event handler PerFrame processes the input render queue. It does two things. One is to generate the loading queue. The other is to pass a copy of the loading queue to the loading thread.

The member function GenerateLoadingQueue does the first thing. This function selects the to-be-loaded out-of-entities of the out-of-core elements from the input render queue in the order of feature importance. The importance of a feature is evaluated by its contribution to the final rendering result. The selected entities are added into the loading queue.

The event handler BeforeFirstFrame is for starting the loading thread, and the event handler AfterLastFrame is for stopping the loading thread and clearing the relevant data, the copy of the loading queue and the cache for the most recently

used out-of-core entities.

The overridable functions Start and Stop are used to start and stop the loading thread at runtime respectively.

- *CLoadingThread*

The class CLoadingThread is a wrap of the loading thread. The most important data members are as follows:

```
TMF<IOutOfCoreObject*>   m_LoadingQueue;
TMF<IOutOfCoreObject*>   m_LoadedEntitiesCache;
```

m_LoadingQueue records a copy of the loading queue.

m_ LoadedEntitiesCache represents a cache, storing the most recently used out-of-core entities. IOutOfCoreObject is the base class of various out-of-core entities, and it can be referred to in subsection 3.9.6 “Out-of-Core Entity”. The data members m_UnusedFrameCount and m_UsedFrameIndex are used to discard the least most recently unused entities. Their declarations are listed as follows:

```
class IOutOfCoreObject
{
protected:
    SFInt           m_UnusedFrameCount;
    SFInt           m_UsedFrameIndex;
    …
}
```

m_UnusedFrameCount is used to count how many frames of the out-of-core entity have not been used up to now. m_UsedFrameIndex records the index of the last frame when the out-of-core entity was used.

The function Service is the main function of the loading thread. It is executed automatically soon after the loading thread is started. The pseudo codes are listed as follows:

```
SFInt CLoadingThread:: Service(void)
{
    while(the thread is not cancelled)
    {
        Wait for the signal sent by CopyLoadingQueue, and the waiting
        time is one second at most.
        while(the reserved memory for the cache is not enough)
        {
            Update m_UnusedFrameCount of the out-of-core entities
            in m_LoadedEntitiesCache
            Sort m_LoadedEntitiesCache on m_UnusedFrameCount in
            ascending order.
```

```
            Unload data of the last out-of-core entity in
            m_LoadedFeatureCache;
            Remove the last out-of-core entity from the cache.
        }

        if(m_LoadingQueue is not empty)
        {
            Load data of out-of-core entities from m_LoadingQueue
            Insert the out-of-core entities into m_LoadedEntitiesCache
        }
    }
    return 1;
}
```

The function Service first waits for the signal that is sent after the function CopyLoadingQueue is executed. Moreover, the longest waiting time is one second. Afterwards, it checks if the memory space of the cache has run out. If so, the least recently unused out-of-core entities are discarded from the cache. Next, it loads data according to the loading queue and puts it into the cache.

The COoCEntityLoader-typed pre-render can be used together with other pre-renders. The following codes show how to adapt to the pre-render CVFCullingPreRender:

```
CVFCullingPreRender* pVFC;
…
IPreRender* pOoCEntityLoader;
pOoCEntityLoader =m_PreRenderMgr.Give("OoCCache", COoCEntityLoader");
pOoCEntityLoader->SetInCamera(m_CameraMgr["Prime"]);
pOoCEntityLoader->SetInRenderQueue (0, pVFC->GetOutRenderQueue[0]);
pOoCEntityLoader->SetInRenderQueue (1, pVFC->GetOutRenderQueue [1]);
pOoCEntityLoader->SetInRenderQueue (2, pVFC->GetOutRenderQueue [2]);
pVFC->SetNextRCU (pOoCCachePreRender);
```

The COoCEntityLoader-typed pre-render COoCEntityLoader is used as the next render control unit of the CVFCullingPreRender-typed pre-render, and takes the output render queues of CVFCullingPreRender-typed pre-render as its input render queues.

3.20.4 CFeatureTypeClassifier

The class CFeatureTypeClassifier represents a kind of pre-render, which traverses the scene and classifies all visited features into several output render queues. Each render queue records one kind of feature.

In most cases, such classification is only required to be done once. Thus, the class CFeatureTypeClassifier only overrides the event handler BeforeFirstFrame.

```
class CFeatureTypeClassifier :public IPreRender
{
friend class CSceneGraphIterator;
protected:
    TMF<SFInt>          m_EntityTypes;
    SFInt               m_FeatureIndex;
    TMF<SFMatrix44f>    m_Matrixes;
public:
    virtual void BeforeFirstFrame(CFrameInfo * pFI);
    SetInEntityTypes(const TMF<SFInt>& EntityTypes, SFInt FeatureIndex);
    static int OnEntry(ISceneIterator * pIterator, void * pParam);
}
```

Users of CFeatureTypeClassifier use the member function SetInEntityTypes to set the desired entity types, and need to guarantee that there are the same number of output render queues as that of desired entity types. The second parameter FeatureIndex of SetInEntityTypes specifies the index of the target feature in m_FeatureSet of the CSGLeaf-typed scene graph leaf node.

The event handler BeforeFirstFrame uses a CSceneGraphIterator-typed iterator to traverse a scene, and provides the callback function CFeatureTypeClassifier:: OnEntry for the iterator to do the classification.

```
void CFeatureTypeClassifier::BeforeFirstFrame(CFrameInfo * pFI)
{
    Clear all output render queues;
    Clear m_Matrixes;

    CSceneGraphIterator iter(m_pSceneModelMpr);
    iter.m_lpfuncOnEntry = CFeatureTypeClassifier::OnEntryNode;

    iter.Create(m_pInTraverseRoot);
    iter.First(m_pInTraverseRoot, this);
    while(!iterator.IsDone())
        iterator.Next(this);
}

int CFeatureTypeClassifier::OnEntry(ISceneIterator*pIterator,void
* pParam)
{
    CSGLeaf* pLeaf = dyanmic_cast< CSGLeaf*>(pIterator->CurrentItem());
    if(pLeaf == NULL)    return 0;
```

```
    CFeatureTypeClassifier* pClassifier = (CFeatureTypeClassifier*)
    pParam;
    const TMF<SFInt>& EntityTypes =  pClassifier->m_EntityTypes;
    SFInt    FeatureIndex = pClassifier->m_FeatureIndex;
    TMF<SFMatrix44f>& Matrixes = pClassifier->m_Matrixes;

    IFeature * pFeature = pLeaf->GetFeatureSet[FeatureIndex];
    if(pFeature->EntityType() is equal to i-th element in EntityTypes)
    {
        Calculate the matrix transforming local coordinate frame to
        world coordinate frame.
        Matrixes.PushBack(matrix);
        m_OutRenderQueues[i]->PushBack(
            CRenderQueueElement(Matrixes.Back(), pFeature));
    }
}
```

The member function OnEntry checks whether the entity type of the feature pointed to by pLeaf->GetFeatureSet[FeatureIndex] is equal to one of the to-be-classified entity types. If it is equal to pClassifier->m_EntityTypes[i], the corresponding feature is placed in the *i*-th output render queue.

The data member CFeatureTypeClassifier::m_Matrixes records matrixes which are used to transform the features from their local coordinate frame to the world coordinate frame.

3.20.5 CRenderQueueElementProcessor

CRenderQueueElementProcessor processes the input render queues and converts each render queue element to a more friendly form for rendering. For example, the default implementation of CRenderQueueElementProcessor can convert CStaticLOD, CBillboard, CSuperShape or CComposedShape features in render queue elements into standard CShape instances. There is no need to set output render queues because CRenderQueueElementProcessor directly modifies the input render queues. Therefore, developers should remember after calling CRenderQueueElementProcessor::PerFrame, all of the input render queues are changed.

By CRenderQueueElementProcessor, the number of distinct render modules can be reduced greatly. Developers can derive subclasses from CRenderQueueElementProcessor to perform more conversions.

The peseudocodes for CRenderQueueElementProcessor are listed as follows:

```
class CRenderQueueElementProcessor : public IPreRender
{
```

```
public:
    virtual void PerFrame(CFrameInfo * pFI);

protected:
    virtual void ProcessElement(CRenderElement* pElement,
                                CRenderQueue* pRenderQueue);
    TMF<SFMatrix44f> m_Matrixes;
}

void CRenderQueueElementProcessor::PerFrame(CFrameInfo * pFI)
{
    m_Matrixes.Clear();
    for(each render queue in m_InRenderQueues)
        for(each element in the render queue)
            ProcessRenderQueueElement(element,render queue);
}

void CRenderQueueElementProcessor:: ProcessElement (CRenderElement*
pElement,CRenderQueue* pRenderQueue)
{
    IFeature* pFeature = pElement->m_pFeature;
    switch(pFeature ->EntityType())
    case Entype_CBillboard:
    {
        Calculate Mrot, the rotation matrix of the billboard according
        to the current camera *m_pInCamera and the rotation axis of
        the billboard pFeature->m_AxisOfRotation;

        SFMatrix44f M = *(pElement->m_pMatrix)* Mrot;
        m_Matrixes.PushBack(M);
        pElement->m_ pMatrix = m_Matrixes.Back();
        break;
    }
    case Entype_CStaticLOD:
    {
        IChildFeature * pSubtreeRoot;
        Select the proper level of detail in the CStaticLOD feature
        according to current camera *m_pInCamera, and let pSubtreeRoot
        point to the level;

        Traverse the subtree whose root node is pointed by pSubtreeRoot,
        and generate a render queue LocalRenderQueue. Each element
        in LocalRenderQueue contains an IShapeFeature instance in
        the subtree, and a matrix that transforms the IShapeFeature
        instance to the coordinate frame of the subtree root node;
```

```
        for each element E_local in LocalRenderQueue
        {
            ProcessElement(&E_local, &LocalRenderQueue);
            *E_local.m_pMatrix = *(pElement->m_pMatrix) * (*E_local.m_pMatrix);
        }

        Replace *pElement with the elements in LocalRenderQueue;
        break;
    }
    case EnType_CSuperShape:
    {
        Generate a render queue LocalRenderQueue, where the i-th
        element contains pFeature->m_Subshape[i] and the corresponding
        matrix;
        Replace *pElement with the elements in LocalRenderQueue;
        break;
    }
    case EnType_CComposedShape:
// Similar to the case of EnType_CStaticLOD, except let pSubtreeRoot
equal pFeature;
        ...
        break;
    default:
        break;
    }
}
```

ProcessRenderQueueElement can be overridden to provide more conversion ability. In the case of converting CStaticLOD or CComposedShape features, ProcessElement is invoked recursively because the class types of the features contained in LocalRenderQueue can be any derived concrete subclasses of IChildFeature.

3.20.6 CLightCullingPreRender

To efficiently support dynamic lighting, we introduce the pre-render CLightCullingPreRender. For each feature, the pre-render finds the lights which directly illuminate the feature. The pre-render firstly select the valid lights, the illumination scopes of which are either intersected with or contained inside the current view frustum. Then, for each valid light, the pre-render determines which features in input rendering queues will be illuminated by the light, and records the pointer to this valid light in the attribute of each to-be-illuminated feature. The

attribute will be used by render modules.

The first input render queue records lights, while the other input render queues records various to-be-rendered features. Different types of lights have different illumination scopes. For example, a point light has a spherical illumination scope, and a spotlight has a conical illumination scope.

Here, the pre-render adopts a visibility culling technique to determine whether a light is valid and whether a feature is illuminated by a given light. It is similar to that used by CVFCullingPreRender.

The declaration and some important function implementations of CLightCullingPreRender are listed as follows:

```
class CLightCullingPreRender: public CVFCullingPreRender
{
public:
    virtual void PerFrame(const CFrameInfo* pFI)
};

void CLightCullingPreRender::PerFrame(const CFrameInfo* pFI)
{
    Clear light pointers recorded in the attributes of all render
    queue elements;
    for each light in the first input render queue
    {
        Calculate the illumination scope of the light;
        If (the illumination scope has non-empty common part with
        the current view frustum)
        {
            For each element in the other input render queues
            {
                If(the element's feature is partly or completely
                inside the illumination scope)
                {
                    Get the attribute of the feature;
                    record the pointer to the light to the attribute;
                }
            } // end for each element
        }
    } // end for each light
}
```

3.21 Examples of Modular Render Pipeline and Render Module

We first introduce an example of a modular render pipeline, CShapeRenderPipeline, and then present an example of a render module, CShapeRenderModule, which is the most important render module in CShapeRenderPipeline.

3.21.1 CShapeRenderPipeline

The class CShapeRenderPipeline is a concrete class of IModularRenderPipeline. CShapeRenderPipeline uses the class CShapeRenderModule, which will be introduced in the next subsection, to draw CShape and CBillboard instances.

Since the OpenGL states need to be set differently for rendering opaque and translucent objects, CShapeRenderPipeline overrides the function DrawRenderQueue to set the proper OpenGL states for drawing opaque and translucent queues.

The declaration and the implementation of CShapeRenderPipeline are as follows:

```
class CShapeRenderPipeline:public IModularRenderPipeline
{
public:
    virtual IRenderModule * MatchRenderModule(IFeature*) ;
    virtual void DrawRenderQueue (CFrameInfo *pFI, SFInt QueueIndex);
}

IRenderModule * CShapeRenderPipeline::MatchRenderModule(IFeature*
    pFeature)
{
    IRenderModuleManager* pRenderModuleMgr;
    pRenderModuleMgr = m_pRenderPipelineMgr->GetRenderModuleManager();
    If(pFeature->EntityType() == Entype_CShape ||
        pFeature->EntityType == Entype_CBillboard)
    {
        CShapeRenderModule* pRenderModule;
        pRenderModule = pRenderModuleMgr->Give("ShapeRM",
            "CShapeRenderModule");
        return pRenderModule;
    }

    return NULL;
}

void CShapeRenderPipeline::DrawRenderQueue (CFrameInfo * pFI, SFInt
    QueueIndex)
{
    // The first input render queue is an opaque queue.
    if(QueueIndex == 0)
    {
        glDisable(GL_BLEND);
        glDisable(GL_ALPHA_TEST);
    }
```

```
    // The second input render queue is a translucent queue
    else if(QueueIndex == 1)
    {
        glEnable(GL_BLEND);
    }

    IModularRenderPipeline:: DrawRenderQueue(pFI, QueueIndex);
}
```

3.21.2 CShapeRenderModule

The class CShapeRenderModule is the most useful render module. CShapeRenderModule is merely for rendering a CShape feature, not for CSuperShape. Such a design is reasonable since the other types of features can be converted into multiple CShape features by the aforementioned pre-render CRenderQueueElementProcessor.

CShapeRMAttribute is the corresponding attribute of class CShapeRenderModule.

```
class CShapeRMAttribute:public IRenderModuleAttribute
{
Public:
    TMF<IGPUResource*>              m_pGPUTextures;
    CBufferGPUResource*             m_pGPUGeometry;

    TMF<CCgProgramGPUResource*>     m_ShaderPrograms;
    TMF<TMF<SFVariant> >        m_ShaderParamSources;
    SFBool                          m_bNeedGPUResources;

    TMF<ILightFeature*>             m_Lights;
}

class CShapeRenderModule:IRenderModule
{
public:
    virtual IRenderModule * Clone() const;
    virtual  IRenderModuleAttribute  *  CreateAttribute(IFeature  *
    pFeature);

    virtual void EstimateCost(
    const CCamera*, const CRenderQueueElement *, SFTime& TimeCost);

    virtual void ApplyForGPUResources(
    const CFrameInfo*, const CCamera *, const CRenderQueueElement *,
    IRenderModuleAttribute * pAttrib,
```

```
    TMF<IFeature*>& GPUResourceApplicants);

    virtual SFBool PrepareResources
    (const CFrameInfo*, const CCamera *, const CRenderQueueElement *,
    IRenderModuleAttribute * pAttrib,
    const TMF<IGPUResource*>& GPUResources);

    virtual void Draw
    (CFrameInfo*pFI, CCamera * pCamera, CRenderQueueElement * pElement,
    IRenderModuelAttribute * pAttrib);
protected:
    IFeature * Check(IGPUResource* pGPUResource, IFeature * pApplicant);
}
```

The class CShapeRenderModule overrides six interface functions of IRenderModule.

```
IRenderModule* CShapeRenderModule::Clone()const;
{
    return new CShapeRenderModule;
}

IRenderModuleAttribute * CShapeRenderModule::CreateAttribute(IFeature *
pFeature)
{
    return new CShapeRMAttribute;
}
```

The implementations of Clone and CreateAttribute are rather simple, which directly return the instance of CShapeRenderModule and CShapeRMAttribute respectively.

3.21.2.1 EstimateCost

The function EstimateCost first checks whether the to-be-rendered feature has changed or not and whether the time cost for rendering the feature has been measured. The time is stored in the relevant attribute. If the feature has not changed and the time has been measured, the function just returns the measured time cost. Otherwise, the time cost should be estimated for this time rendering.

The hardware processing capability and the complexity of the to-be-rendered feature should be considered in estimating the rendering time cost. The complexity of the feature is usually measured by the number of triangles and texels. The time cost is the sum of the time for preparing GPU resources and that for drawing.

```
void CShapeRenderModule::EstimateCost(CCamera*,
     CRenderQueueElement * pElement,
     IRenderModuleAttribute * pAttrib,
     SFTime& TimeCost);
{
    if(The corresponding feature has been modified && pAttrib->
    GetTimeCost()>0)
    {
        TimeCost = pAttrib->GetTimeCost();
    }else{
        Based on current hardware configuration, estimate TimeCost,
        the time for GPU resource preparing and drawing according
        to the face number, vertex number and pixels number.
    }
}
```

3.21.2.2 ApplyForGPUResources

The function ApplyForGPUResources first examines whether relevant GPU resources should be applied for, so as to render this feature in the current frame. It is done by checking the flag m_bNeedGPUResources of the input attribute. This flag is also maintained by this function. If the GPU resources have been allocated and the feature has not changed since the allocation, the flag is set to be true.

If the resources are required, the function applies for the corresponding GPU resources for geometry, textures and shaders. If texture GPU resources are applied for, the function first checks whether the texture is single texture or multiple textures.

The render module uses CShapeRMAttribute::m_GPUTextures and CShapeRMAttribute:: m_ShaderPrograms to record the allocated texture GPU resources and shader program GPU resources when the applications are successful, which will be seen in the function PrepareResources. Therefore, the arrays of CShapeRMAttribute::m_GPUTextures and CShapeRMAttribute::m_ShaderPrograms are resized to be the same as the number of the corresponding applicants.

The function Check, which appears in ApplyForGPUResources for many times, is to check whether the GPU resources have ever been allocated for an applicant. If they have been allocated, the function Check will return NULL.

```
void CShapeRenderModule::ApplyForGPUResources(CFrameInfo*, CCamera *,
CRenderQueueElement * pElement, IRenderModuleAttribute * pAttrib,
TMF<IFeature*>& GPUResourceApplicants)
{
    CShapeRMAttribute* pShapeAttribute = (CShapeRMAttribute*) pAttrib;
    If(the corresponding feature has been modified)
        pShapeAttribute->m_bNeedGPUResources = true;
```

```
    else if(!pShapeAttribute->m_bNeedGPUResources)
        return;

    CShape* pShape = (CShape *)pElement->m_pFeature;

    // Apply for buffer GPU resource
    IFeature*pApplicant;
    pApplicant = Check (pAttrib->m_pGPUGeometry,
        pShape->GetGetGeometry().p);
    GPUResourceApplicants.Add(pApplicant);

    // Apply for texture GPU resources
    ITextureFeature* pTexture = pShape->GetAppearance()->GetTexture().p;
    if(pTexture points to a CMultiTexture object)
    {
        for(i=0; i< pTexture->GetTextures().size() ++i)
        {
            pApplicant=Check(pAttrib->m_GPUTextures[i],
            pTexture->GetTextures()[i].p);
            GPUResourceApplicants.Add(pApplicant);
        }

    pAttrib->m_GPUTextures.Resize(pTexture->GetTextures().size());
    }else{
        pApplicant = Check(pAttrib->m_GPUTextures[0], pTexture);
    }

// Apply for shader program GPU resources
    Let pActiveShader point to the active shade in the appearance object
    *pAppearance;
    TMF<TAddress<IShaderProgram>>&
    Programs=pActiveShader->GetShaderPrograms();
    for(i= 0; i < Programs.size(); ++i)
    {
        pApplicant = Check(pAttrib->m_ShaderPrograms[i], Programs[i].p);
        GPUResourceApplicants.Add(pApplicant);
    }
    pAttrib->m_ShaderPrograms.Resize(Programs.size());
}

IFeature * CShapeRenderModule::Check(IGPUResource* pGPUResource,
IFeature * pApplicant)
{
    return pGPUResource == NULL ? pApplicant : NULL
}
```

3.21.2.3 PrepareResources

The function PrepareResources first checks the flag m_bNeedGPUResources of the input attribute. This check is consistent with that in ApplyForGPUResources. It means that if there is no need to apply for GPU resources, there is no need to prepare these resources.

The main tasks performed by PrepareResources are as follows.

(1) It records the pointers to allocated GPU resources in the attribute to facilitate the use of these resources.

(2) It sets the data sources of the shader program parameters. There are two kinds of shader program parameters. One is the system variable, and the other is user-defined constants. For a shader program, any system variable is denoted by a string-typed name. Then, this function uses the variable name to find the address of the relevant variable, and wraps the address with an SFVariant instance. The SFVariant instance is regarded as the data source of the shader program parameter and it is saved in CShapeRMAttribute:: m_ShaderParamSources. Since any user-defined constant is represented by an SFNamedVariant instance in a shader program, the function copies the value to an SFVariant instance and records it in CShapeRMAttribute:: m_ShaderParamSources too.

In fact, no one can guarantee that the applied GPU resources are always available. Therefore, the function PrepareResources makes a check in the end to see whether the pointers to the GPU resources recorded in the attribute are all valid. If all GPU resources are ready, the function sets the value of pAttrib->m_bNeedGPUResources to be false, and returns true value. Otherwise, it returns false value, which means the shape is not drawn.

```
SFBool CShapeRenderModule::PrepareResources(CFrameInfo*,
CCamera *, CRenderQueueElement * pElement, IRenderModuleAttribute * pA,
const TMF<IGPUResource*>& GPUResources)
{
    CShapeRMAttribute* pAttrib = (CShapeRMAttribute *) pA;

    If(!pAttrib->m_bNeedGPUResources )
        return;

    CShape* pShape = (CShape *)pElement->m_pFeature;
    SFInt index = 0;

// Record the pointers of the allocated GPU resources in the attribute
// *pAttrib.
    if(pAttrib->m_pGPUGeometry == NULL){
        pAttrib->m_pGPUGeometry = GPUResources[index++];
    }
    for(i=0; i<pAttrib->m_GPUTextures.size(); ++i)
    {
```

```
        if(pAttrib->m_GPUTextures[i] == NULL)
            pAttrib->m_GPUTextures[i] = GPUResources[index++];
    }
    for(i=0; i<pAttrib->m_GPUShaderPrograms.size(); ++i)
    {
        if(pAttrib->m_GPUShaderPrograms[i] == NULL)
            pAttrib->m_GPUShaderPrograms[i] = GPUResources[index++];
    }

// Find the sources of shader parameters and record them in attribute
// Let pActiveShader point to the active shade in the appearance object
// *pAppearance;
    TMF<TAddress<IShaderProgram> >& Programs=pActiveShader->
    GetShaderPrograms();
    for(i= 0; i < Programs.size(); ++i)
    {
        for(each variable name in Program[i].m_SystemVariables)
        {
            Find the system variable according to the variable name;
            Let pSysVar point to the system variable;
            Create an SFVariant instance to wrap pSysVar, as SFVariant
            * pVar = new SFVariant(pSysVar);
            pAttrib->m_ShaderParamSources[i].Add(pVar);
        }

        for(each user defined constant in
            Program[i].m_UserDefinedConstants)
        {
            Create an SFVariant instance pointed by pVar to wrap the
            user defined constant;
            pAttrib->m_ShaderParamSources[i].Add(pVar);
        }
    }

    If(pAttrib->m_pGPUGeometry is not NULL, and
       every pointer in pAttrib->m_GPUTextures is not NULL, and
       every pointer in pAttrib->m_GPUShaderPrograms is not NULL)
    {
        pAttrib->m_bNeedGPUResources = false;
        return true;
    }else{
        return false;
    }
}
```

3.21.2.4 Draw

The function Draw has two processing branches. If the to-be-rendered shape is without any shader program, the function DrawViaFixedPipeline is invoked. Otherwise, the function DrawViaProgrammablePipeline is invoked.

```
void CShapeRenderModule::Draw(CFrameInfo* pFI,
     CCamera * pCamera,
     CRenderQueueElement * pElement,
     IRenderModuelAttribute * pAttrib)
{
    Check the pAttribute->m_ShaderPrograms.size()
    If(there is no shader program)
        DrawViaFixedPipeline(pFI,pCamera, pElement,pAttrib);
    else
        DrawViaProgrammablePipeline(pFI,pCamera, pElement, pAttrib);
}
```

The function DrawViaFixedPipeline makes proper settings for OpenGL context, and binds allocated resources, such as textures, vertex buffer and index buffer. It supports the following rendering features:

(1) Dynamic local illumination;
(2) Alpha culling and alpha blending;
(3) Multiple texture mapping;
(4) Environmental mapping.

```
void CShapeRenderModule::DrawViaFixedPipeline(CFrameInfo* pFI,
CCamera * pCamera, CRenderQueueElement * pElement, IRenderModuleAttribute
* pAttrib)
{
    CShape* pShape = (CShape *) pElement->m_pFeature;
    CAppearance* pAppearance = (CAppearance*) pShape ->GetAppearance();
    CMaterial* pMaterial = (CMaterial*)pAppearance->GetMaterial();

// Enable blending mode if the material is transparent
    if(pMaterial->GetTransparency() > 0)
    {
        glEnable(GL_Blend);
        glBlendFunc(GL_ONE, GL_ONE_MINS_SRC_ALPHA);
    }

// Enable alpha culling if the alpha clamp is set;
    If(pMaterial-> GetAlphaCullingThreshold() >0.0)
    {
        Enable alpha test function
```

```
        Set alpha test reference;
    }

// Enable lighting and set light parameters
    if((CShapeRMAttribute *)pAttrib->m_Lights.size() > 0)
    {
        glEnable(GL_LIGHTING);
        for each light in pAttrib->m_Lights
            Set light parameters in OpenGL context;
    }

// Set color and material
    glColorv(pMaterial->GetDiffuse());
    Set material according to pMaterial by calling a serial of glMaterial(...);

// Set textures
    ITextureEntity* pTexture = pAppearance->GetTexture().p;
    CShapeRMAttribute * pShapeAttrib = (CShapeRMAttribute*)pAttrib;
    TMF<CTexGPUResource*>& GPUTextures= pShapeAttrib-> m_pGPUTextures;


    if(pTexture points to a single texture feature)
    {
        Set the texture's filters and wrap mode;
        Set the texture's transform matrix;
        if(the texture is environmental texture)
            Generate texture coordinates;
        GPUTextures[0]->Bind();
    }else{// pTexture points to a multiple texture feature
        for(every texture in the multiple texture feature)
        {
            Set the texture's filters and wrap mode;
            Set current texture stage modulate mode
            GPUTextures[i]->Bind();
        }
    }

// Set geometry
    pShapeAttribute->m_pGPUGeometry->Bind();

// Draw elements
    SFInt Count = pShape->GetGeometry()->GetIndex().size();
    SFInt Type = m_pGPUGeometry->m_IndexComponentType;
    glDrawElements(GL_TRIANGLES,Count,Type,0);
```

```
// Unbind GPU resources
    for(i = 0; i < pShapeAttrib->m_GPUTextures.size();++i)
        pShapeAttrib->m_GPUTextures[i]->Unbind();
    pShapeAttri->m_pGPUGeometry->Unbind();
}
```

The function DrawViaProgrammablePipeline exploits Cg context to execute Cg shader programs.

```
void CShapeRenderModule:: DrawCGFragmentProgram (CFrameInfo*pFI,
CCamera * pCamera,
CRenderQueueElement * pElement,
IRenderModuelAttribute * pAttrib,
SFInt& CPUMemoryCostInThisFrame)
{
    CShape* pShape = (CShape *) pElement ->m_pFeature;
    CShapeRMAttribute* pShapeAttrib = (CShapeRMAttribute*) pAttrib;
    CMaterial* pMaterial = (CMaterial*)pAppearance->GetMaterial();

    // Enable blend if the material is transparent
    if(pMaterial->GetTransparency() > 0.0)
    {
        glEnable(GL_Blend);
        glBlendFunc(GL_ONE, GL_ONE_MINS_SRC_ALPHA);
    }

    // Enable if the material alpha clamp is set;
    If(pMaterial->GetAlphaCullingThreshold() >0.0)
    {
        Enable alpha test function;
        Set alpha test reference;
    }

    if(the shape has no fragment shader program)
    {
        Setmaterialandtexturesinthesimilarwaytothatinfixedpipeline;
    }

    // Set geometry
    pShapeAttrib->m_pGPUGeometry->Bind();

    // Set shader parameters
    for(i=0; i< pShapeAttrib->m_Shaders.size();++i)
    {
        pShapeAttrib->m_ShaderPrograms.Bind(pShapeAttrib->
        m_ShaderParamSources[i]);
```

```
    }

    // Draw elements
    SFInt Count = pShape->GetGeometry()->GetIndex().size();
    SFInt Type = m_pGPUGeometry->m_IndexComponentType;
    glDrawElements(GL_TRIANGLES,Count,Type,0);

    // Unbind GPU resources
    pShapeAttrib->m_pGPUGeometry->Unbind();
    for(i =0; i < pShapeAttrib->m_ShaderPrograms.size()++i)
        pShapeAttrib->m_ShaderPrograms.Unbind();
    if(the shape has no fragment shader program)
        Unbind textures
}
```

The following is an example of Cg fragment shader program. This fragment shader program uses a diffuse map, light map, specular map, normal map and reflection map to represent the shading effects of the surface. The color of point *v* on the surface is computed by the following formular.

$$\mathrm{Color}(v) = \mathrm{DiffuseComponent}(v) + \mathrm{SpecularComponent}(v) + \mathrm{ReflectiveComponent}(v)$$

$$\mathrm{DiffuseComponent}(v) = \mathrm{DiffuseMap}(\mathrm{TexCoord0}(v)) * \mathrm{LightMap}(\mathrm{TexCoord1}(v))$$

$$\mathrm{SpecularComponent}(v) = \sum_i \mathrm{SpecularMap}(\mathrm{TexCoord2}(v)) * \mathrm{Light}_i * \mathrm{Alpah}_i^{\mathrm{shiness}}$$

$$\mathrm{ReflectiveComponent}(v) = \mathrm{Reflectance} * \mathrm{RelectionMap}(\mathrm{TexCoord3}(v, \mathrm{viewpoint}))$$

TexCoord[0|1|2|3] represents the texture mappings for different textures respectively. Since ReflectionMap represents environmental mapping, the texture mapping TexCoord3 is view-dependent. Alpha_i is the angle between the reflected ray on point *v* shot from Light_i and the direction from point v to the viewpoint. LightMap records radiance on the surface and it is the computation result of global illumination. DiffuseMap and SpecularMap depict the diffuse and specular coefficients of the material.

The following are the Cg pseudo codes:

```
pixout main(vertout IN,
       uniform sampler2D Shape_DiffuseMap,
       uniform sampler2D Shape_LightMap,
       uniform sampler2D Shape_SpecularMap,
       uniform sampler2D Shape_NormalMap,
       uniform sampler2D Shape_ReflectionMap,
       uniform float Shape_Shininess,
       uniform float Shape_Reflectance,
       uniform float System_Lights_Count,
       uniform float3 System_Lights_Posistion[8],
       uniform float4 System_EyePosition)
```

```
{
    pixout OUT;
    float4 diffuse;
    float4 reflect;
    float4 specular;

    Calculate DiffuseComponent;
    Calculate SpecularComponent;
    Calculate ReflectiveComponent;

    OUT.color=DiffuseComponent+SpecularComponent+ReflectiveComponent;
    return OUT;
}
```

Here all parameters are system variables. Readers can refer to the part about "IShaderFeature" in subsection 3.5.10 "IAppearanceFeature and Related Features". The value of these system variables are updated per frame through the corresponding data sources. The following code in the function Draw binds the data sources to the shader program parameters:

```
    for(i=0; i< pShapeAttrib->m_Shaders.size();++i)
    {
        pShapeAttrib->m_ShaderPrograms.Bind(pShapeAttrib->
        m_ShaderParamSources[i]);
    }
```

In fact, what shader programs are used to render a feature is specified by the shape itself, since the to-be-used shader programs are just stored in the shape's appearance. CShapeRenderModule merely provides a framework for applying shader programs.

3.22 Implementation Details of CRenderingEngine

This section unveils the implementation details of some important member functions of CRenderingEngine.

3.22.1 Configure

The function Configure is used to configure the composed part inside a CRenderingEngine instance. Developers can input a configure file for this function, or override CRenderingEngine::OnConfigure to customize the configuration of a rendering engine.

Of course, CRenderingEngine provides a default configuration. The default configuration mainly does the following things:

(1) Registers the prototypes of IPreRender and IRenderPipeline concrete classes. If some modular render pipelines are adopted, the related render module prototypes will be registered during the render pipeline prototype registration.

(2) Generates the "Prime" camera and render target. The prime camera is the camera that is set by CRenderingEngine::SetCamera. The prime render target is the render window, corresponding to the window specified by CRenderingEngine::m_WinInfo.

(3) Generates three render queues with the names of "Opaque", "Translucent" and "Reflective". The name indicates the material characteristics of the entities in the queues. For example, the "Opaque" queue contains the opaque entities.

(4) Generates a render queue for lights with the name of "Lights".

(5) Generates the "Prime" render pipeline. Sets the "Prime" camera and the "Opaque", "Translucent" and "Reflective" render queues as its inputs, while setting the "Prime" render target as its output.

(6) Generates four pre-renders, such as "Classifier", "VFCulling", "RQEProc", "LightCulling", whose types are CFeatureTypeClassifier, CVFCullingPreRender, CRenderQueueElementProcessor and CLightCullingPreRender respectively. The pre-render "Classifier" selects various lights from the scene graph to form the render queue "Lights". The pre-render "VFCulling" does the view frustum calling during the scene traverse with respect to the "Prime" camera, and classifies the possibly visible entities into three render queues as its outputs. The pre-render "RQEProc" does some preparation for the to-be-rendered features, and decomposes some composed features into several CShape instances. The pre-render "LightCulling" selects proper lights for each to-be-rendered shape.

(7) Forms the control flow by connecting the pre-renders and the render pipeline.

The pseudo codes are listed as follows:

```
int CRenderingEngine::Configure(const SFString& ConfigFileName)
{
    m_PreRenderMgr.SetRenderingEngine(this);
    m_RenderPipelineMgr.SetRenderingEngine(this);

    if(!ConfigFileName.empty())
        return ConfigureByScript(ConfigFileName);

    if(OnConfigure())
        return 1;

    RegisterPreRenderPrototypes();
    RegisterRenderPipelinePrototypes();

// Prime camera
```

```
    m_CameraMgr.Give("Prime");
// Prime render target
    m_RenderTargetMgr.Give("Prime");

// Render queues for opaque, translucent and reflective features.
    CRenderQueue* Q[3];
    Q[0] = m_RenderQueueMgr.Give("Opaque");
    Q[1] =m_RenderQueueMgr.Give("Translucent");
    Q[2] =m_RenderQueueMgr.Give("Reflective");
    Q[3] =m_RenderQueueMgr.Give("Lights");

// Default prime render pipeline
    CShapeRenderPipeline * pRndPL;
    pRndPL = m_RenderPipelineMgr.Give("Prime", "CShapeRenderPipeline");
    pRndPL->SetInCamera(m_CameraMgr["Prime"]);
    pRndPL->SetInRenderQueue(0, Q[0]);
    pRndPL->SetInRenderQueue(1, Q[1]);
    pRndPL->SetInRenderQueue(2, Q[2]);
    pRndPL->SetOutRenderTarget(m_RenderTargetMgr["Prime"]);
    pRndPL->SetBudget(RenderBudget);

// Default pre-renders
    pPreRender0=m_PreRenderMgr.Give("Classifier","CFeatureTypeClassifier");
    pPreRender1 = m_PreRenderMgr.Give("VFCulling", "CVFCullingPreRender");
    pPreRender2=m_PreRenderMgr.Give("RQEProc","CRenderQueueElementProcessor");
    pPreRender3=m_PreRenderMgr.Give("LightCulling","CLightCullingPreRender");

// Set the pre-render "Classifier", which will find all lights in the
// first scene graph
    TMF<SFInt> Types;
    Types.Add(EnType_CSpotLight);
    Types.Add(EnType_CPointLight);
    Types.Add(EnType_CDirectionalLight);
    pPreRender0->SetEntityTypes(Types, 0);
    pPreRender0->SetOutRenderQueue(0, Q[3]);
    pPreRender0->SetOutRenderQueue(1, Q[3]);
    pPreRender0->SetOutRenderQueue(2, Q[3]);

// Set the pre-render "VFCulling", which will perform view frustum culling
    pPreRender1->SetInCamera(m_CameraMgr["Prime"]);
    pPreRender1->SetOutRenderQueue(0, Q[0]);
    pPreRender1->SetOutRenderQueue(1, Q[1]);
    pPreRender1->SetOutRenderQueue(2, Q[2]);

// Set the pre-render "RQEProc", which will update the orientation
// of CBillboard instances, select proper detail level of CStaticLOD
```

```
// instance, and decompose each CSuperShape and CComposedShape feature
// into several CShape features.
// No need to set any output render queue, because the pre-render will
// modify the input render queue directly.
    pPreRender2->SetInCamera(m_CameraMgr["Prime"]);
    pPreRender2->SetInRenderQueue(0,Q[0]);
    pPreRender2->SetInRenderQueue(1,Q[1]);
    pPreRender2->SetInRenderQueue(2,Q[2]);

// Set the pre-render "LightCulling", which will select proper lights for
// each to-be-rendered feature.
    pPreRender3->SetInCamera(m_CameraMgr["Prime"]);
    pPreRender3->SetInRenderQueue(0,Q[3]); // The render queue "Lights"
    pPreRender3->SetInRenderQueue(1,Q[0]);
    pPreRender3->SetInRenderQueue(2,Q[1]);
    pPreRender3->SetInRenderQueue(3,Q[2]);

// Connect pre-renders and the render pipeline
    pPreRender0->SetNextRCU(pPreRender1);
    pPreRender1->SetNextRCU(pPreRender2);
    pPreRender2->SetNextRCU(pPreRender3);
    pPreRender3->SetNextRCU(pRndPL);

// Set the pre-render "Classifier" to be the start pre-render.
    m_pStartPreRender = pPreRender0;

    return 1;
}
```

3.22.2 Initialize

The function Initialize is to create the OpenGL rendering context, set the viewport of the "Prime" camera and create the "prime" render target.

The pseudo codes are listed as follows:

```
int CRenderingEngine::Initialize(const CWinInfo& WinInfo)
{
    m_WinInfo = WinInfo;
    Create the OpenGL rendering context, m_WinInfo.m_hRC if it is NULL;
    Set the viewport of m_CameraMgr["Prime"] equal to the window's client
    region;
    m_RenderTargetMgr["Prime"].CreateRenderWindow(m_WinInfo);
}
```

3.22.3 DoRendering

The function DoRendering is to update frame information and to execute m_pStartPreRender->_DoPerFrame.

```
void CRenderingEngine::DoRendering(CFrameInfo * pFI)
{
    Update the frame information *pFI;
    m_pStartPreRender->_DoPerFrame(pFI);
}
```

3.22.4 OpenSceneModel

The function OpenSceneModel mainly does the following things:

(1) It uses the scene model manipulator to open a scene model.

(2) If the KD-tree of the active scene graph has not been built yet, it builds the tree.

(3) It sets the root of the active scene graph's KD-tree to be the render traverse root. It means this KD-tree will be traversed and rendered. Besides, every pre-render's traverse root is the root of the active scene graph's KD-tree in default. If the traverse root of some pre-render cannot be set like this, developers can reset the traverse root in the pre-render'event BeforeFirstFrame.

(4) It specifies the viewing parameters of the prime camera, m_CameraMgr["Prime"]. If there exist some predefined viewpoints in the scene model, the prime camera's viewing parameters are set to be equal to the first viewpoint. If there is no predefined viewpoint, it computes the prime camera's viewing parameters so that the entire scene, represented by the active scene graph, can be viewed.

(5) It activates the event BeforeFirstFrame of the start pre-render.

```
int CRenderingEngine::OpenSceneModel(const SFString& ModelName, int
ActiveSceneGraph)
{
    if(!m_pSceneModelMpr->OpenSM(ModelName))
        return 0;

    if(NULL == m_pSceneModelMpr->GetSI(ActiveSceneGraph))
        Build KD tree of the first scene graph;

    SetRenderTraverseRoot(m_pSceneModelMpr->GetSI(ActiveSceneGraph));
    for each *pPreRender in m_PreRenderMgr
        pPreRender->SetInTraverseRoot(pRoot);

    Specify the viewing parameters of m_CameraMgr["Prime"] according
```

```
    to the active scene graph.

    CFrameInfo FrameInfo;
    Initialize FrameInfo;
    m_pStartPreRender->BeforeFirstFrame(&FrameInfo);

    return 1;
}
```

3.23 Conclusion

This chapter describes in detail a specific framework for a real-time rendering engine. Instead of exploring the algorithm and the implementation (in later chapters of this book we will do so), we have tried to show how to design a reasonable framework for a real-time rendering engine. The so-called reasonableness, we believe, relies on the following two points:

(1) Comprehensibility. For either engine developers or programmers who build applications based upon the engine, a comprehensible framework will make it easy for them to keep the program model in mind, which greatly reduces the workload of programming and debugging, alleviates the difficulty of learning and decreases the probability of making mistakes. Moreover, obviously, another advantage of the framework's comprehensibility is that it is easy to maintain the engine codes. Since the design ideas of the framework are understandable, those people who maintain the engine can easily follow the concept of the designers and developers.

(2) Extensibility. The framework should have sufficient flexibility for multi-level extensions. This sounds easy, but it is rather difficult in practice. Designers should predict future demands to leave enough space for these extensions. However, a real-time rendering engine is, after all, a system that requires high performance. Excessive extensible space in the design will cause unnecessary performance degradation. It requires designers to strike a careful balance between application demands and performance.

For a real-time rendering system design, there are many factors that need to be considered. However, with comprehensibility and extensibility, the engine has a solid basis for success and the remaining work is to choose appropriate algorithms to enrich it.

Honestly speaking, designing a "good" real-time rendering engine is quite a complex thing, which may require significant experiments and iterative design processes. If readers want to have their own rendering engine, the first step is to make full preparation knowing that it requires tremendous passion and perseverance to complete a real-time rendering engine. We hope this chapter will be useful.

References

Department of Army (1973) Universal Transverse Mercator Grid. U. S. Army Technical Manual TM 5-241-8: 64

Angel E (2005) Interactive Computer Graphics: A Top-Down Approach Using OpenGL (4th Edition). Addison Wesley Publishing, Boston

Brutzman D, Daly L (2007) X3D: Extensible 3D Graphics for Web Authors. Morgan Kaufmann Publishers, Massachusetts

Fernando R, Kilgard MJ (2003) The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics. Addison Wesley Publishing, Boston

Gamma E, Helm R, Johnson R, Vlissides JM (1994) Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Publishing, Boston

Khronos Group (2007) OpenGL SDK. http://www.opengl.org/sdk

OpenGL Architecture Review Board, Shreiner D, Woo M, Neider J, Davis T (2007) OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2.1 (6th Edition). Addison Wesley Publishing, Boston

Pilone D, Pitman N (2005) UML 2.0 in a Nutshell. O'Reilly Media Publishing, Sebastopol

Web3D Consortium (2003) ISO/IEC 14772:1997 Virtual Reality Modeling Language. http://www.web3d.org/x3d/

Web3D Consortium (2008) ISO/IEC 19775-1.2:2008 X3D Architecture and base components Edition 2. http://www.web3d.org/x3d/

4

Rendering System for Multichannel Display

Through combining or stitching a tiled display, multichannel display technology provides us with an affordable, customizable and flexible way to implement a large-scale display of ultra high resolution. There have been more and more applications in scientific visualization, industrial design, military simulation, digital entertainment, business presentations and other fields.

For the physical composition, a display channel corresponds to a display device, which can be an LCD monitor, a projector, or other display equipment. A display device is connected to a graphics card's video output port. In terms of graphics display, the contents of the video output of the graphics card are a sequence of images generated by a graphics rendering channel. As a number of physical display devices should logically form one large-scale display system, each graphics rendering channel produces only one part of the whole image, which is to be shown by the large-scale display system. Therefore, the images generated by each rendering channel must ensure the following criteria.

(1) The spatial continuity of the generated images. That is, the contents of the images shown in the adjacent display screens must be continuous. In other words, the contents can be stitched and formed into a picture with geometric continuity.

(2) The time synchronization of the rendering channels. All rendering channels must generate and output corresponding sub-images of one whole image synchronously with the same pace.

These rendering channels are combined into a parallel rendering system. Therefore, the parallel rendering system is the core of the multichannel display.

4.1 The Overview of Parallel Rendering

In the parallel rendering system, each channel is a rendering pipeline. And a rendering pipeline primarily consists of geometry and rasterization modules. There exists some kind of transversal network across all paralleled rendering pipelines. Through the transversal network, to-be-drawn objects, or say graphic primitives,

and the synchronization signals are passed across these pipelines, achieving rendering synchronization and the rational allocation of computing resources.

According to the relative position of the transversal network along the rendering pipelines, Molnar S *et al*. (Molnar, 1994) classified parallel rendering systems into three categories: sort-first, sort-middle and sort-last.

In any sort-first system, the transversal network works in the front-end of rendering pipelines. Each pipeline corresponds to a sub-region of the entire display area. First of all, the to-be-drawn objects are processed by a certain algorithm, such as a view frustum culling algorithm. Then they are sent to the proper rendering pipelines and drawn by the pipelines respectively. Inside the rendering pipelines, there are no more transversal networks and no transmission of to-be-drawn objects. The sort-first systems are easy to implement, since no special hardware is required to be developed. It can be built on the basis of a computer cluster system with a high-speed local area network. The drawback is that it is rather difficult for a sort-first system to realize load balancing, which requires extra treatment.

In any sort-middle system, the transversal network works between geometry processing and rasterization modules in rendering pipelines. Each rendering pipeline's rasterization module corresponds to the sub-region of the entire display area. According to some load balancing strategy, all input graphic primitives are sorted and sent to relevant pipelines to perform geometry processing. When these graphic primitives are converted to screen coordinates, they are reassigned to rendering pipelines through the transversal network to do rasterization. This method is easy for realizing the load balancing of geometry processing. Nevertheless, it requires a high-performance transversal network between geometry processing and rasterization modules across rendering pipelines. Therefore, this method can be implemented only by specially-designed hardware, such as SGI's InfiniteReality system (Montrym *et al*., 1997). It is extremely difficult to achieve with a cluster of workstations.

In any sort-last system, the transversal network works at the end of the rasterization stage. In a sort-last system, all input graphic primitives are assigned to different pipelines to do geometry processing and rasterization according to some load balancing strategy. The output of each rendering pipeline is a depth image, which is sent through the transversal network to a depth image composition node. Based on pixels' coordinates and the depth of these depth images, the hidden pixels are removed and all of the depth images are synthesized into one image. Then, according to the viewing region of the display devices, the final image is split into tiles, each of which is to be displayed by one display device. It is easy for this kind of system to implement load balancing. However, it requires special hardware to implement high-performance depth image transferring, synthesizing and splitting. In particular when the required image resolution is much higher, the hardware requirements become quite stringent.

With the rapid development of high-speed networks and 3D graphics technology, it has become a hot research topic to build a parallel rendering system on a computer cluster with a high speed network, because of its attractive performance-

price ratio and good scalability. Therefore, the sort-first parallel rendering systems have gradually become popular. In a computer cluster, a rendering channel is typically performed by a computer, which is called a render node. Its rendering output corresponds to a sub-region display area. In addition, there is usually a computer as a control node, which is used to assign a collection of graphic primitives to rendering nodes.

The sort-first system is further classified into client-server and master-slave classes, according to the data distribution form and the responsibility of the cluster nodes (Staadt *et al*., 2003).

4.1.1 Client-Server

In a client-server system, the render nodes are regarded as servers that provide various render services. These nodes are also called render servers. The control node is regarded as the client who calls the services provided by the render servers. However, in most cases the control node is actually a controller and manages all rendering servers. The render controller, the software running on the control node, takes responsibility of distributing graphic primitives from the control node to render servers and synchronizes these render servers. Therefore, user applications based on a client-server system should be deployed on the control node. The applications call all rendering functions provided by the render controller either through across-processes or across-thread communications. The render controller generates rendering commands and relevant graphic primitives according to the calls invoked by user applications. Since each render server has a view frustum, the render controller distributes these rendering commands and graphic primitives to appropriate render servers according to the spatial relationship between these primitives and the viewing frustums. On the other side, each render server accepts the rendering command and graphic primitives, and executes the rendering commands with the relevant graphic primitives. Besides, the render controller provides some mechanism to synchronize all render servers. One popular way is for the render controller to broadcast rendering commands and each render server executes the commands once it receives them.

We can further classify the client-server systems into the immediate mode and the retained mode according to the storage position of graphic primitives. For a client-server system running in the immediate mode, all graphic primitives are stored in the render controller and some of the primitives are sent to render servers for each frame. At the same time, for each frame, the render server will discard the graphic primitives once they are drawn. The advantage of the immediate mode is that it can be implemented in low-level graphics libraries such as OpenGL. Therefore, it is trivial to port an application from a single computer to a computer cluster. However, the drawback of this mode is that it requires much network bandwidth for data transfer, because graphic primitives are transferred per frame. Caching graphic primitives in render servers is an effective way to reduce the

consumption of the network bandwidth.

For systems running in the retained mode, all graphic primitives are transferred from render controller to render servers once before performing rendering. Compared with the immediate mode, the retained mode does not require the transfer of a large amount of graphic primitives during rendering. Therefore, its network bandwidth consumption is much less than the immediate mode, and the overall performance is higher for the same hardware configuration. The drawback is that the dataset of graphic primitives should be duplicated for multiple render servers. It is rather complicated to keep the consistence of all these dataset copies. It is not an easy process to port an application from a single computer to a computer cluster.

4.1.2 Master-Slave

In the master-slave system, the render node is called slave, and the control node is called master. The application software is deployed in every slave. That means each slave runs an instance of application software. The master guarantees the consistence of the running states of these application instances. The master usually takes charge of handling user interactions. Besides user inputs, all factors that may influence applications' running states, such as timestamps and random numbers, must be controlled by the master. During runtime, only messages about these influence factors are transferred from the master to the slaves, so the network bandwidth is little required. Since the application developer must be aware of these influence factors and must follow a certain framework to make sure they are carefully handled, the freedom of application development is somehow restricted.

4.2 The Architecture of a Cluster-Based Rendering System

First of all, to distinguish the single computer rendering system and the cluster-based render system, we address the former system as a rendering engine and the rear one as a cluster-based rendering system.

From this section, we will introduce a prototype system of parallel rendering, the Visionix Cluster-based Rendering System (VCRS), which was developed by ourselves. It is built upon the basis of the rendering engine introduced in Chapter 3. In this chapter, this rendering engine is called a Visionix Rendering Engine (VRE). Through the introduction of the prototype system, we hope that readers will have some idea about developing a distributed parallel rendering system based on an ordinary rendering engine.

The VCRS is a sort-first client-server system, and runs in retained mode. Like most client-server systems, VCRS has two parts, the render controller and the

render server. As the names indicate, the render server runs on the render node and the render controller runs on the control node. The user applications are built upon the render controller and deployed on the control node.

The render controller mainly consists of three modules, the rendering system interface, the server manager and the rendering engine:

(1) The rendering system interface is the VCRS' façade, encapsulating all necessary system services to application programs.

(2) The server manager is designed to fulfill two tasks. One task is to translate local service requests, which are called by applications, to network messages defined in the rendering protocol, and send these messages to render servers. The other task is to manage all render servers, including synchronizing their rendering process, keeping data consistent, providing a synchronization clock, etc.

(3) The rendering engine is just the aforementioned VRE, which is designed for a single computer.

In the render controller, the rendering engine manages a scene model. The rendering system interface uses the rendering engine to access the scene model, and uses the server manager to transfer the scene model data to render servers.

The render server mainly consists of the server interface and the rendering engine. The server interface is designed to communicate with the render controller, such as to receive and send various messages. The rendering engine is the Visionix Rendering Engine, which performs various rendering services. The overall structure of the VCRS is illustrated in Fig. 4.1.

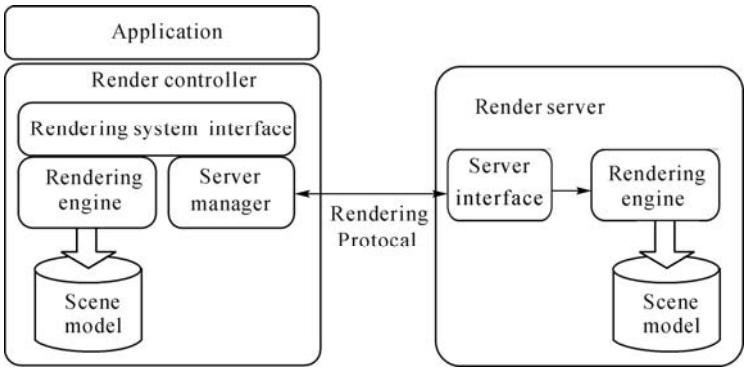

Fig. 4.1 The architecture of the Visionix Cluster-based Rendering System (VCRS)

4.3 Rendering System Interface

Through the rendering system interface, the cluster-based rendering system could be used as a rendering system for a single computer. All networking operations are transparent to application developers.

The class vxIRenderingSystem is the main interface for this rendering system, providing services such as start-up and shut-down system, do rendering per frame, etc. In addition, it aggregates two interfaces represented by the class vxIModel and vxIUI. The class vxIModel is for scene model management, and the class vxIUI is for user interface management.

Through the class vxIModel and some relevant interface classes, various scene entities can be manipulated. The interface classes for the scene entities are listed as follows:

(1) vxISceneGraph, vxISGNode. The class vxISceneGraph represents the interface of a scene graph and vxISGNode is the interface for any nodes inside the scene graph.

(2) vxIShape. It represents various renderable objects, which act as basic bricks of a virtual world. It is equivalent to the class IShapeFeature in the VRE (see subsection 3.5.5). It provides the interfaces vxIGeometry and vxIAppearance for developers to manipulate the geometry and appearance of the shape feature. The interface vxIGeometry describes a variety of geometries, such as point, line, sphere, triangle set, elevation grid, etc. The interface vxIAppearance further provides the interface vxITexture for texture operations.

(3) vxIAnimation. It represents animated objects, and is equivalent to the class IAnimatedFeature in the VRE (see subsection 3.5.6).

(4) vxILight. It represents various artificial lights, such as street lamps, reading lamps, pharos and searchlights and so on. It is equivalent to the class ILightFeature in the VRE (see subsection 3.5.7).

(5) vxIEnvironment. It is designed to depict visual effects of the environment, such as sky, clouds, sun, moon, fog and rain.

Through the class vxIUI, various user interface elements can be manipulated. The classes for the UI elements are listed as follows.

(1) vxIScreen. It is the interface to represent display screens. A vxIScreen instance could represent either one LCD monitor, or a tiled display system consisting of many projectors.

(2) vxIWindow. It represents windows shown on screens.

(3) vxIViewport. It represents viewports, the regions inside windows. Each viewport shows a certain view of a scene and corresponds to a camera.

(4) vxICamera. It is the interface of cameras to view scenes.

(5) vxIUIControl. It is the interfaces of various user interface controls, such as button, list, check box, etc.

The following codes show how to create an instance of vxIRenderingSystem. Here we use the design pattern factory:

```
vxCRenderingSystemFactory factory;
vxIRenderingSystem System = factory.Create();
```

The reader may find that the function vxRenderingSystemFactory::Create returns a vxIRenderingSystem instance instead of its pointer or reference. The

reason is that here we adopt the smart reference technique. Any instance of class where the name is prefixed with vxI-, such as vxIRenderingSystem, vxIModel, vxIUI, vxISceneGraph, vxISGNode and vxIShape, represents an interface object and can be regarded as a smart reference. The lifetime of entities referenced by these interface objects are managed by the rendering system framework. Any unused entity would be collected in a garbage bin and the garbage bin would be cleaned later by the rendering system. Therefore, developers are not concerned with the memory release issues of unused entities.

4.3.1 vxIRenderingSystem

In this section, we will briefly explain the main public member functions of class vxIRenderingSystem. We roughly classify the member functions into several groups. In each group, we select several of the most useful functions to explain.

- *Components*

```
vxIModel  GetModel();
vxIUI     GetUI();
```

The two functions are used to get the aggregated interface objects, vxIModel and vxIUI.

- *Runnin_g*

```
SFInt     StartUp(const SFString& ConfigFileName);
Void      ShutDown();
```

These functions control the running states of the rendering system. Any other member functions should be invoked after calling StartUp and before calling ShutDown. The input parameter ConfigFileName of the function StartUp is a file path to a configuration file, which specifies the configuration of the entire system. If it fails to make the configuration or another error occurs during the startup, StartUp returns zero or negative values. Otherwise, it returns one.

The configuration file defines the network, user interface, input devices and audio settings. The network settings include the render controller and render server's message listen and sent ports. By using this port information, either the render controller or the render servers could find each other by broadcasting some messages.

The user interfaces settings are rather complicated. The main part of the user interface describes the screen configuration. For each screen, a number of the screen's properties are specified. The screen properties are organized into several sections as follows.

Basic. It contains the usage description, screen position in the display space, screen type, screen size, screen resolution, etc.

Windows. This section describes parameters for creating windows and their relevant UI controls. These parameters also can be written in one or more separate files, and these files' URL are placed in this section. This kind of file is referred to as UI resource file.

Stereo configuration. It contains the stereo mode (active or passive), the distance between left and right eyes, and the two render server groups for the left eye and the right eye if the passive stereo mode is enabled.

Rendering option. It contains anti-alias option, render target resolution, animation options, seed for random number generator, maximum time to set up GPU resources per frame, cache size in GPU memory, minimum time for object staying in host memory, and so on.

Render server. It contains the information about all render servers that do rendering for this screen, such as render context creation parameters, output image correction settings for tiled display, and so on.

The input devices settings specify which devices are used and relevant properties. The audio settings include the default audio volume, acoustic field settings, 3D sound settings, and so on.

```
void      DoRendering();
void      SwapBuffer();
```

The function DoRendering draws contents into the rendering buffers for all displays once. The function SwapBuffer swaps the back and front rendering buffers once.

- *Rendering options*

```
void      SetRenderingErrorUpperBound(SFInt bound);
void      SetRenderingRateLowerBound(SFInt bound);
```

This first function sets the rendering error upper bound, which is used to select proper level-of-details. For any object with levels of details, the shape difference between the rendering results of this object's coarse model version and that of the finest model version is defined as the image-space error of the coarse model version. The upper bound of this image-space error adaptively controls the fineness of scene models. If a smaller upper bound were specified, the finer model versions of objects in the scene would be selected to be rendered. Therefore, the smaller upper bound yields the higher rendering quality.

The second function sets the lower bound for rendering rate. When the lower bound is specified, it means the rendering system is optimized for time efficiency and attempts to guarantee the rendering rate greater than the lower bound.

The rendering rate lower bound is of higher priority than the rendering error upper bound if both of them are set. In other words, if the rendering time buffer

defined by the rendering rate lower bound were not enough, the rendering system would increase the rendering error upper bound to reduce the fineness of to-be-rendered objects or, in other words, reduce the model complexity. Therefore, this system tends to trade rendering quality for rendering speed.

- *Garbage cleaning*

```
void CleanGarbage ();
```

Any scene model or user interface entities that have been created by the rendering system but have not been used are considered as a piece of garbage. This function collects all garbage and cleans it to release more memory.

4.3.2 vxIModel

The class vxIModel is a façade of the entire scene model or, in other words, scene database. It provides all kinds of manipulating operations for the scene database. Compared with the class CSceneModelManipulator2 discussed in Chapter 3, we simplify the public member functions of the class vxIModel so that it is easier for application developers to use it.

A model represented by an instance of the class vxIModel usually consists of one or more scene graphs. Each scene graph usually has one or more layers, and each layer is a graph where each node contains one or more scene features. The VCRS provides the class vxISceneGraph and vxISGNode for developers to manipulate the graph structure of a scene model. As its name indicates, the class vxISceneGraph represents scene graphs. The class vxISGNode represents layers, middle nodes and leaf nodes inside layers.

To lessen the efforts to learn and use the VCRS, we simplify the interface classes of scene features. So far, the VCRS has defined four interface classes, as vxIShape, vxIAnimation, vxIEnvironment and vxILight. The class vxIShape is similar to the class IShapeFeature (see subsection 3.5.5), representing all kinds of features that have geometry and appearance. The class vxIAnimation corresponds to the class IAnimatedFeature. The class vxIEnvironment represents various environmental effects, such as fog, rain, cloud, sky, sun and so on. The vxILight is similar to the class ILightFeature (see subsection 3.5.7), representing various lights.

In the following, we will introduce the public member functions of the class vxIModel.

- *Scene model management*

```
SFBool CreateSceneModel(const SFString& name,
             const SFString& sg0_name, const SFString& si0_name);
```

```
SFByte OpenSceneModel(const SFString& name, SFInt mode);
SFBool CloseSceneModel();
SFBool IsOpened() const;
```

The first three functions are designed to create, open and close a scene model database respectively. The function IsOpened checks whether the model is opened. These functions are similar to those of CSceneModelManipulator2 in subsection 3.10.1.

- *Scene graph management*

```
SFInt          AddNewSceneGraph(const SFString& name="");
SFBool         RemoveSceneGraph(SFInt index);
vxISceneGraph  GetSceneGraph(SFInt index);
SFInt          GetSceneGraphCount() const;
SFInt          GetSceneGraphIndex(const SFString& name);
SFBool         Rename(SFInt index, const SFString& new_name);
```

The function AddNewSceneGraph is for creating a new scene graph with the input name and adding it into this scene model. The return value is the index of the newly added scene graph in the scene model. If the index is negative, it means it fails to create or add the layer.

The function RemoveSceneGraph removes the scene graph from the scene model according to its index. When a scene graph is removed from the scene model, the scene graph becomes an unused one and it will be released during the garbage clean done by the system.

The function GetSceneGraph is for obtaining the scene graph interface from its index, and the function GetSceneGraphCount is for obtaining the number of scene graphs in this scene model. The scene graph index can be obtained by the function GetSceneGraphIndex according to its name. The function Rename changes the scene graph name into the input string variable new_name.

- *Persistent scene feature creation*

```
vxILight       CreateLight(SFInt32 type, const SFString& name="");
vxIShape       CreateShape(SFInt32 type, const SFString& name="");
vxIGeometry    CreateGeometry(SFInt type, const SFString& name="");
vxIAppearance  CreateAppearance(SFInt32 type, const SFString& name="");
vxIAnimation   CreateAnimation(SFInt32 type, const SFString& name="");
vxITexture     CreateTexture(SFInt32 type, const SFString& name="");
vxIEnvironment CreateEnvironment(SFInt32 type, const SFString&
               name="");
```

This group of functions is designed to create various scene features and return their interfaces. Here we follow the term feature, which is introduced in Section 3.3 “Basics of Scene Model”. Once a scene feature is created by one of these

creation functions, the newly-born feature is just in an unused state until it is added in one scene graph somewhere.

The first parameter specifies the type of the concrete feature. Table 4.1 shows the valid values for the type and their corresponding feature classes defined in VRE. Such correspondence can be explained through the following example. If a vxILight instance is created with the type value equal to EnType_CPointLight, it means this instance represents the interface of a CPointLight feature. These scene feature interfaces will be introduced in subsections 4.3.2.2 – 4.3.2.5.

Table 4.1 The valid values for the type and their corresponding feature classes defined in VRE

Function	Valid values for the type	Feature classes in the VRE
CreateLight	EnType_CPointLight, EnType_CDirectionLight, EnType_CSpotLight	CPointLight, CDirectionLight, CSpotLight
CreateShape	EnType_CShape, EnType_CSuperShape, EnType_CBillboard.	CShape, CSuperShape, CBillboard
CreateGeometry	EnType_CPoint, EnType_CLine, EnType_CPolyline, EnType_CIndexedTriangleSet.	CPoint, CLine, CPolyline, CIndexedTriangleSet
CreateAppearance	EnType_CPointAppearance, EnType_CLineAppearance, EnType_CAppearance.	CPointAppearance, CLineAppearance, CAppearance.
CreateAnimation	EnType_CSkinAnim	CSkinAnim
CreateTexture	EnType_CImageTexture	CImageTexture
CreateEnvironment	EnType_CEnvironment	CEnvironment

All features created by these functions are persistent. If a feature is persistent, it means the feature will be created in the host memory and saved in the database. Otherwise, it is only created in the host memory and not stored in the database. The temporal features are used to represent points, lines, planes, bounding boxes and other items shown in visual tools. In vxISceneGraph, we define a group of functions to manipulate such kinds of temporal features (see subsection 4.3.2.1).

- *Transaction*

```
void    BeginTransaction();
SFBool  RollbackTransaction();
SFBool  CommitTransaction();
```

These functions are designed to begin, rollback and commit a transaction to the model or, in other words, the database. They are similar to those defined in CSceneModelManipulator2 (see subsection 3.10.1). All creation, removal and modification of persistent features should be done inside a transaction.

- *Import*

```
void  ImportScene (const SFString& file_path, vxISGNode parent);
```

This function imports a scene represented in Virtual Reality Modeling Language from an external file into this model, and stores it in the database. The second parameter parent specifies a scene graph node, which will be the parent node of the imported scene.

4.3.2.1 vxISceneGraph and vxISGNode

- *vxISceneGraph*

The class vxISceneGraph is the interface of scene graph objects. The corresponding class in the VRE is CSceneGraph. The public member functions of vxISceneGraph are classified into several groups.

- *Layer*

```
SFInt      AddNewLayer(const SFString& name);
SFBool     RemoveLayer(SFInt index);
vxISGNode  GetLayer(SFInt Index) const;
SFInt      GetLayerCount() const;
SFInt      GetLayerIndex(const SFString& name) const;
SFBool     RenameLayer(SFInt index, const SFString& new_name) const;
```

The function AddNewLayer creates a new layer with the input name, and adds it to this scene graph. The return value is the index of the newly-added layer in the scene graph. If the index is negative, it means it fails to create or add.

The function RemoveLayer removes a layer from the scene graph according to its index. When a layer is removed from the scene graph, the layer becomes an unused one in the scene model and it will be released by the garbage clean of the scene model.

The function GetLayer is used to obtain the layer interface according to its index, and the function GetLayerCount is to obtain the number of layers in this scene graph. The layer index can be obtained by the function GetLayerIndex according to the layer name.

- *Temporal scene feature*

This group of functions is designed to manipulate temporal scene features. The temporal scene features will be released automatically after the scene model is closed. The following functions can be used outside the transaction.

```
SFInt  AddNewTempLight(SFInt32 type, const SFString& name="");
SFInt  AddNewTempShape(SFInt32 type, const SFString& name="");
SFInt  AddNewTempGeometry(SFInt type, const SFString& name="");
SFInt  AddNewTempAppearance(SFInt32 type, const SFString& name="");
SFInt  AddNewTempAnimation(SFInt32 type, const SFString& name="");
SFInt  AddNewTempTexture(SFInt32 type, const SFString& name="");
SFInt  AddNewTempEnvironment(SFInt32 type,const SFString& name="");
```

The above functions create a variety of temporal features and add them to the scene graph temporarily. The index of the newly-born temporal feature in the scene graph is returned. The first parameter type is the same as that in those functions for creating a persistent scene feature (see Table 4.1).

```
void            RemoveTempFeature(SFInt index);
```

This function removes the specified temporal feature from the scene graph.

```
vxILight        GetTempLight(SFInt index);
vxIShape        GetTempShape(SFInt index);
vxIGeometry     GetTempGeometry(SFInt index);
vxIAppearance   GetTempAppearance(SFInt index);
vxIAnimation    GetTempAnimation(SFInt index);
vxITexture      GetTempTexture(SFInt index);
vxIEnvironment  GetTempEnvironment(SFInt index);
```

Each of the above functions obtains the relevant interfaces of the features specified by its index.

```
SFInt           GetTempFeatureCount(SFInt interface_type);
SFInt           GetTempFeatureIndex(SFInt interface_type, const
                SFString& name);
```

The function GetTempFeatureCount returns the number of the features which have the same interface type specified by the input parameter interface_type. The value of the parameter interface_type could be IType_Light, IType_Shape, IType_Geometry, IType_IAppearance, IType_Animation, IType_Texture or IType_Environment.

The function GetTempFeatureIndex returns a feature's index according to the input interface type and name.

- *Query*

```
vxISGNode   FindNode(const SFString& name) const;
```

The function FindNode finds a scene graph middle or leaf node according to

its name.

```
SFBool    QueryByRay(TMF<vxISGNode>& query_results,
          TMF<SFVec3d>& intersection,
          const SFVec3d& ray_origin, const SFVec3d& ray_direction,
          vxISGNode subgraph_root, SFUInt mode);
```

This function queries the scene graph leaf nodes whose bounding boxes are intersected by a ray in the given subscene graph specified by the parameter subgraph_root. If the value of the subgraph_root is invalid, the entire scene graph is queried. Besides, the intersection points are output. The value of the parameter mode could be QUERY_ALL_INTERSECTION or QUERY_FIRST_INTERSECTION. If the query mode is QUERY_ALL_INTERSECTION, all intersected scene graph leaf nodes are collected in the output array query_results. If the mode is QUERY_ FIRST_INTERSECTION, only the firstly intersected leaf node is collected.

```
SFBool  QueryByExtrudedVolume(TMF<vxISGNode>& query_results,
        const MFVec3d& base_polygon, const SFVec3d& extrusion_vector,
        vxISGNode subgraph_root, SFUInt mode);
```

The function QueryByExtrudedVolume finds the scene graph leaf nodes whose bounding boxes have a specified spatial relation with the given extruded volume and in the given scene graph. The last parameter mode specified the spatial relation. It could be one or some combination of the following values.

QUERY_INSIDE. It means the expected leaf nodes' bounding boxes must be completely inside the volume.

QUERY_INTERSECT. It means the expected leaf nodes' bounding boxes must intersect the volume.

QUERY_ENCLOSE. It means the expected leaf nodes' bounding boxes must enclose the volume.

QUERY_OUTSIDE. It means the expected leaf nodes' bounding boxes must be completely outside the volume.

If the combination QUERY_INSIDE_VOLUME | QUERY_INTERSECT_ VOLUME is specified, it means that at least some part of the expected leaf nodes' bounding boxes must be inside the volume.

The extruded volume is represented by a base polygon and an extrusion vector. The extrusion vector represents the direction and length of the extrusion.

```
SFBool  QueryByPolygonOnViewport(TMF<vxISGNode>& query_results,
        const MFVec3d& polygon, const vxIViewport& viewport,
        vxISGNode subgraph_root, SFUInt mode);
```

This function finds the scene graph leaf nodes whose bounding boxes' projection on the viewport is of a specified spatial relation with the input polygon. The valid value of the last parameter mode is the same as that of the parameter mode in

function QueryByExtrudedVolume. However, this mode specifies the 2D spatial relation between the bounding boxes' projection and the given polygon, rather than the 3D spatial relation between the bounding boxes and extruded volume in QueryByExtrudedVolume.

```
SFBool  QueryBySphere(TMF< vxISGNode >& query_results,
        const SFVec3d& sphere_center, SFDouble sphere_radius,
        vxISGNode subgraph_root, SFUInt mode);
```

The function QueryBySphere is similar to the function QueryByExtrudedVolume except this function uses sphere instead of extruded volume to calculate the spatial relation.

- *Shadow*

```
SFInt        AddNewShadowRegion(const SFBox& region, const
             SFString& name="");
void         RemoveShadowRegion(SFInt index);
void         GetShadowRegionCount() const;
const SFBox& GetShadowRegion(SFInt index) const;
SFInt        GetShadowRegionIndex(const SFString& name) const;
SFBool       RenameShadowRegion(const SFString& new_name) const;
```

The above functions manage several shadow regions. When a shadow region is defined in this scene graph, it means that all shadow in this region should be computed. In other words, all scene features in the shadow regions become a shadow receiver.

```
void         SetShadowCaster(const TMF< vxISGNode >& node_array);
```

The function SetShadowCaster sets an array of scene graph nodes as shadow casters.

- *Spatial index*

```
SFBool    BuildSpatialIndex(SFInt KeyShapeSubscript, SFInt32
          MaxShapeCountInLeaf);
void      SetSpatialIndexUpdatingMode(SFInt KeyShapeSubscript,
          SFInt Mode);
```

The function BuildSpatialIndex builds a spatial index of this scene graph. Because our scene graph representation allows multiple shapes in one leaf node (see Section 3.6), it should specify which shape in the leaf node should be used as the "keyword", or key shape, to build the spatial index. The first input parameter KeyShapeSubscript is a subscript value of the shape arrays in the leaf nodes for specifying the key shape. One key shape can be used to build one spatial index at

most. Therefore, a key shape subscript can be used as an identifier for its corresponding spatial index of a certain scene graph. Moreover, if there has been one spatial index built on a key shape, it is also valid to invoke the function BuildSpatialIndex for the same key shape. It is equivalent to updating the spatial index to reflect the changes in its relevant scene graph. The second parameter MaxShapeCountInLeaf is for specifying the maximum number of shapes in the leaf nodes of the spatial index.

The function SetSpatialIndexUpdatingMode tells the system how to update a spatial index once its relevant scene graph has been changed. There are two updating modes, the immediate mode and the deferred mode. In the immediate mode, any change of scene graph will incur the update of its spatial index, and the updating is transparent to developers. In the deferred mode, regardless what changes occur in its scene graph, the spatial index will not be updated until developers explicitly invoke the function BuildSpatialIndex.

- *vxISGNode*

The class vxISGNode is the interface of the various scene graph nodes, which are represented by CSGLeaf, CSGDynLeaf, CSGComposition, CSGDynComposition, CSwitch and CLayer in the VRE. The public member functions of vxISGNode are listed as follows.

- *Type and name*

```
SFInt             GetType() const;
const SFString&   GetName() const;
SFBool            Rename(const SFString& new_name);
```

The first two functions are designed to obtain node type and name respectively. The function Rename assigns a new name to the node.

- *Transform*

```
void    GetLocalTransform(SFTransform& transform);
void    SetLocalTransform(const SFTransform& transform);
void    GetWorldTransform(SFTransform& transform);
```

The first two functions are used to get and set the local transform, which is the transform of this node's coordinate frame relative to its parent node's coordinate frame. The third function calculates the world transform, which can directly transform coordinates in this node's coordinate frame into the scene graph's coordinate frame, namely the world coordinate frame.

- *Hierarchy*

```
SFInt      AddNewChild(SFInt type, const SFString& name="");
```

```
void        RemoveChild(SFInt index);
SFInt       GetChildrenCount() const;
vxISGNode   GetChild(SFInt index);
```

The function AddNewChild creates a scene graph node of specified type and adds it to this node. The first parameter type indicates the scene graph node type. The valid value of this parameter could be EnType_CSGComposite, EnType_CSGDynComposite, EnType_CSGSwtich and EnType_CSGLeaf or EnType_CSGDynLeaf. These types correspond to the scene graph node class CSGComposite, CSGDynComposite, CSGSwitch, CSGLeaf and CSGDynLeaf. These classes can be referenced in Section 3.6 “Scene Graph”. The second parameter of these functions is the name of to-be-created vxISGNode instances. The name for scene graph nodes could be empty. Nevertheless, if the non-empty name is given, it must be unique in the name space of the entire scene model. The index of a newly-added child node will be returned, and the index can be used by the function GetChild to obtain the child node’s interface.

The function RemoveChild removes the specific child node. The function GetChildrenCount obtains the total of children nodes.

- *Feature*

```
SFInt            AddShape (vxIShape shape);
SFInt            AddAnimation(vxIAnimation animation);
SFInt            AddLight(vxILight light);
SFInt            AddEnvironment(vxIEnvironment environment);
void             RemoveFeature(SFInt index);
vxIShape         GetShape (SFint index) const;
vxIAnimation     GetAnimation(SFInt index) const;
vxILight         GetLight(SFInt index) const;
vxIEnvironment   GetEnvironment(SFInt index) const;
SFInt            GetFeatureType(SFInt index) const;
SFInt            GetFeatureCount() const;
```

The first four functions add relevant features to this node respectively. Each of them returns the index of the newly-added feature in the node. The function RemoveFeature removes the feature specified by the index from the node. Each of the functions GetShape, GetAnimation, GetLight and GetEnvironment returns a feature interface according to its index in the node respectively. If the feature interface specified by the index is not the expected type, the returned interface object is invalid. For example, if the second feature in the node were not a shape feature, the interface returned by GetShape(1) would not be valid. The validation of a feature interface can be done by the interface’s member function IsValid. The function GetFeatureType returns the type of the feature, and the return value can be one of the following values, IType_Shape, IType_Animation, IType_Environment and IType_Light. The last function returns the number of the features in the node.

- *Visibility*

```
SFBool    GetVisibility() const;
void      SetVisibility(SFBool bshow);
```

The two functions set the visibility tag of the node. If a node's visibility is set falsely, any child or features inside the scene graph node will not be rendered.

- *Switch*

```
SFInt     GetWhichChoice() const;
void      SetWhichChoice(SFInt32 value);
```

The two functions take effect only for the nodes of the type EnType_CSwitch. For these switch-typed nodes, at most one child together with this child's descendant nodes would be chosen as the part of the scene graph for rendering. These functions are used to get and set the index of the chosen child. Readers can make reference to the introduction about CSwitch in Section 3.6 "Scene Graph".

4.3.2.2 vxIShape and relevant interfaces

- *vxIShape*

The class vxIShape represents the interface of the feature classes CShape, CSuperShape and CBillboard. The member functions are listed in the following groups.

- *Basic*

```
SFBool    IsValid()  const;
SFInt     GetFeatureType() const;
```

The first function IsValid validates the interface. The function GetFeatureType returns the type of the feature pointed by this interface.

- *Billboard*

```
SFBool    SFVec3f& GetAxisOfRotation() const;
void      SetAxisOfRotation(const SFVec3f& value);
```

The functions are used to get and set the axis of rotation for a billboard feature. If the interface does not point to a billboard feature, the function GetAxisOfRation

returns a zero vector and the function SetAxisOfRotation does nothing.

- *Super shape*

```
SFInt     GetSubShapeCount() const;
vxIShape  GetSubShape(SFInt index);
```

The two functions can only deal with the interface for the feature represented by the class CSuperShape. The first function obtains the number of subshapes. The second function gets the interface of a specific subshape according to its index.

- *Geometry and appearance*

```
vxIGeometry GetGeometry();
vxIAppearance  GetAppearance();
```

These functions get the geometry and appearance interfaces respectively.

- *vxIGeometry*

The class vxIGeometry is the interface of a variety of geometries, such as point, line, sphere.

- *Basic*

```
SFBool    IsValid() const;
SFInt     GetFeatureType() const;
```

These functions are the same as those in the class vxIShape.

- *Point*

```
SFBool    GetPoint(SFVec3d& point) const;
void      SetPoint(const SFVec3d& point);
```

If the feature pointed by this interface is a point, these functions get and set the coordinates of this point respectively. If this interface does not point to a point, the function GetPoint returns a false value and the function SetPoint does nothing.

- *Line*

```
SFBool GetLine(SFVec3d& start_point, SFVec3d& end_point) const;
void   SetLine(const SFVec3d& start_point, const SFVec3d& end_point);
```

These functions get and set the relevant information about a line feature.

- *Polyline*

```
SFBool GetPolyLine(SFBool& closed_flag, MFVec3d& point_array) const;
SFInt  SetPolyLine(SFBool closed_flag, const MFVec3d& point_array);
```

These functions get and set the relevant information about a polyline feature. The first parameter indicates whether this line is closed.

- *Indexed face set*

```
SFBool  GetIndexedFaceSet(MFVec3f& coord, MFVec2f& TexCoord,
            MFVec3f& normal, MFColorRGB& color, MFInt32& index,
            SFFloat& crease_angle, SFUInt32& option) const;
void    SetIndexedFaceSet(const MFVec3f& coord, const MFVec2f&
            TexCoord, const MFVec3f& normal, const MFColorRGB&
            color, const MFInt32& index, SFFloat crease_angle,
            const SFUInt32 option);
void    SetBox(const SFBox& box);
void    SetExtrusion(const MFVec3d& base_polygon,
            const SFVec3d& extrusion_vector);
```

This group of functions is used to get and set information about an indexed face set. The function SetBox sets relevant geometric information for an axis-aligned cuboid, and the function SetExtrusion sets that for an extruded body.

- *vxIAppearance*

The class vxIAppearance is the interface of the appearance feature for point, line and face. The corresponding feature classes are CPointAppearance, CLineAppearance and CAppearance.

- *Basic*

```
SFBool    IsValid()          const;
SFInt     GetFeatureType()   const;
```

These functions are the same as those in the class vxIShape.

- *Point and line appearance*

```
SFBool   GetPointAppearance(SFByte& point_type,SFColorRGB&
             color, SFFloat& thickness);
void     SetPointAppearance(SFByte point_type, const SFColorRGB&
             color, SFFloat thickness);
SFBool   GetLineAppearance(SFByte& line_type, SFColorRGB& color,
             SFFloat& thickness);
```

```
void     SetLineAppearance(SFByte line_type, const SFColorRGB&
              color, SFFloat thickness);
```

These functions get and set the properties for point and line appearance.

- *Face appearance*

This group of functions is valid only for the feature represented by the class CAppearance.

```
SFBool       GetMaterial(CMaterial& material) const;
void         SetMaterial(const CMaterial& material);
vxITexture   GetTexture(SFInt stage) const;
SFInt        SetTexture(SFInt stage, vxITexture texture);
SFBool       GetShaderConstantF(const SFString& register,
                 SFFloat*pdata, SFInt count) const;
SFBool       GetShaderConstantI(const SFString& register,
                 SFInt*pdata, SFInt count) const;
SFBool       GetShaderConstantB(const SFString& register,
                 SFBool*pdata, SFInt count) const;
SFInt        SetShaderConstantF(const SFString& register,
                 SFFloat*pdata, SFInt count);
SFInt        SetShaderConstantI(const SFString& register,
                 SFInt*pdata, SFInt count);
SFInt        SetShaderConstantB(const SFString& register,
                 SFBool*pdata, SFInt count);
SFInt        GetFillingMode() const;
void         SetFillingMode(SFInt mode);
SFInt        GetShadingMode() const;
void         SetShadingMode(SFInt mode);
SFBool       GetHighlightingMode(SFInt& mode, SFInt& type,
                 SFVec3f& param) const;
void         SetHighlightingMode(SFInt mode, SFInt type,
                 const SFVec3f& param);
```

The function GetMaterial and SetMaterial get and set the surface material. The class CMaterial is introduced in subsection 3.5.10. The functions GetTexture and SetTexture get and set a texture at a specific texture stage. The class vxITexture will be introduced in the following subsection. The shader constants can be obtained or set through the function GetShaderConstant or SetShaderConstant. The last six functions are designed to get or set the filling mode, shading mode and highlighting mode. Table 4.2 shows the possible filling modes.

Table 4.2 Filling modes

Filling mode	Meaning
FILLING_ POINT	Only the vertices are drawn
FILLING_ WIREFRAME	Only the edges are drawn
FILLING_ FACE	Only the faces are rasterized. It is the ordinary mode
Any combination of the above values	

The feasible shading modes are listed in Table 4.3.

Table 4.3 Shading modes

Shading mode	Meaning
SHADING_ DEFAULT	The ordinary way is adopted to compute shading
SHADING_ COLOR	Only pre-computed color information stored in the geometry feature is used to compute shading
SHADING_TEXTURE	Only the textures are used to color faces
SHADING_ MATERIAL	Only the material is used to compute shading
SHADING_COLOR_TEXTURE	It is equal to SHADING_COLOR|SHADING_ TEXTURE. Both color and textures are used
SHADING_MATERIAL_TEXTURE	Equal to SHADING_MATERIAL|SHADING_ TEXTURE

The feasible highlighting modes and style are listed in Table 4.4.

Table 4.4 Highlighting modes

Highlight mode	Style	Meaning
HL_NONE		It is not highlighted
HL_ BRIGHTNESS	LINEAR, QUADRIC	It is highlighted by changing brightness. If the style is LINEAR, the new brightness y = b+a*x, where x is the old brightness, b = parameter.vec[0], a = parameter.vec[1]. If the style is QUADRIC, the new brightness y = c+b*x+a*x*x, where x is the old brightness, c = parameter.vec[0], b = parameter.vec[1], a = parameter.vec[2]
HL_ TRANSPARENCY	LINEAR, QUADRIC	It is highlighted by changing transparency. The changing style is similar to that of HL_BRIGHTNESS
HL_ BBOX	CORNER, WIREFRAME	It is highlighted by displaying the relevant bounding box If the style is CORNER, the corner of bounding box is displayed. If the style is WIREFRAME, the wireframe of the bounding box is displayed. The parameter is the color for drawing the bounding box

- *vxITexture*

The class vxITexture is the interface of the class CImageTexture.

```
SFBool IsValid() const;
SFInt  SetTexture(const SFString& image_name, SFBool external_flag);
void   GetTextureFilter(SFInt& type, SFInt& mode) const;
void   SetTextureFilter(SFInt type, SFInt mode);
void   GetTextureWrapMode(SFInt& type, SFInt& mode) const;
void   SetTextureWrapMode(SFInt type, SFInt mode);
```

The function SetTexture sets up a texture either from an external image file or from an existent texture feature in the model. The Boolean parameter external_ flag is used to choose the texture setting fashion. If the parameter external_flag is true, the parameter image_name is a file path of an image. If external_flag is false, image_name is the name of a texture feature.

The texture filter and wrap mode can be obtained or set through the relevant functions listed above.

4.3.2.3 vxIAnimation

The class vxIAnimation is the interface for an animated feature, which is represented by only one class CSkinAnim so far. Application developers can control the animation state through function SetPlayState. The value of the parameter mode can be either PLAYBACK or STOP. If the mode is PLAYBACK, the animation will be played back and the parameter motion_index specifies which motion is played if there are several motions in the animation feature. The parameter start_time determines the starting position of the to-be-played motion. If the mode is STOP, the animation will be stopped and the other parameters are ignored.

After the function GetPlayState is invoked, if the output parameter mode is PLAYBACK, the parameter motion_index and play_time output the index and time of the motion currently played respectively. If the output mode is STOP, motion_index is the index of the stopped motion, and play_time is the time when the motion is stopped.

```
SFBool  IsValid() const;
SFInt   GetFeatureType()  const;
void    SetPlayState(SFInt mode, SFInt motion_index,
            SFTime start_time);
void    GetPlayState(SFInt& mode, SFInt& motion_index,
            SFTime& play_time) const;
```

4.3.2.4 vxILight

The class vxILight is the interface of the feature classes CPointLight, CDirectionLight and CSpotLight. The main public functions are almost the same as those in the feature classes. Therefore, we just list them and do not give a detailed explanation about the meaning of these functions.

- *Basic*

```
SFBool      IsValid() const;
SFInt       GetFeatureType() const;
SFFloat     GetAmbientIntensity() const;
void        SetAmbientIntensity(SFFloat value);
SFColor&    GetColor() const;
void        SetColor(const SFColor&value);
SFFloat     GetIntensity() const;
void        SetIntensity(SFFloat value);
SFBool      IsLightOn() const;
void        LightOn(SFBool value);
```

The above functions can be used for the three kinds of features. The function names just indicate these functions' functionality.

- *Direction light*

```
const SFVec3f&   GetDirection() const;
void             SetDirection(const SFVec3f&value);
```

The functions are valid only for direction lights.

- *Point light and spotlight*

```
SFVec3f&    GetAttenuation() const;
void        SetAttenuation(const SFVec3f&value);
SFVec3f&    GetLocation() const;
void        SetLocation(const SFVec3f&value);
SFFloat     GetRadius() const;
void        SetRadius(SFFloat value);
```

The functions are valid for both point lights and spotlights.

- *Spot light only*

```
SFFloat     GetBeamWidth() const;
void        SetBeamWidth(const SFFloat&value);
SFFloat     GetCutOffAngle() const;
```

```
void        SetCutOffAngle(const SFFloat& value);
```

The functions are valid only for spotlights.

4.3.2.5 vxIEnvironment

The class vxIEnvironment is the interface of the class CEnvironment which represents several environmental effects, such as sky, sun, moon, cloud, fog and rain. In a scene graph, although there could be more than one environmental feature, only one environmental feature will take effect at one time.

- *Basic*

```
SFBool  IsValid() const;
SFBool  IsEffectEnabled(SFInt effect_type) const;
void    EnableEffect(SFInt effect_type, SFBool enable);
void    GetReferenceFrame(SFInt& frame_type, SFVec3d& position,
            SFDatetime& date_time) const;
void    SetReferenceFrame(SFInt frame_type, const SFVec3d& position,
            SFDatetime time);
```

Various effects can be enabled through the function EnableEffect. So far, the supported effect types are EF_FOG, EF_RAIN, EF_CLOUD, EF_SUN, EF_MOON, EF_SKY. The function IsEffectEnabled could tell us whether an effect is enabled.

There are two reference frames for environments. One is the geodetic spatial reference frame for earth. Then the parameter position in the function SetReferenceFrame is the triple (longitude, latitude, altitude), which are the geographic coordinates of the world origin. In this case, the parameter frame_type is RF_EARTH_GD. The parameter date_time specifies the date and time of the environment. The other reference frame type is RF_USER. It means the feature describes an environment at any place at any time. In this case, both the parameter position and data_time are ignored.

- *Effect properties*

The following functions get and set environmental effect properties.

```
void    GetFogProperty(const SFColorRGB& color,
            SFFloat& visibility_distance) const;
void    SetFogProperty(const SFColorRGB& color,
            SFFloat visibility_distance);
void    GetRainProperty(SFFloat& raindrop_density) const;
void    SetRainProperty(SFFloat raindrop_density);
void    GetCloudProperty(SFInt& cloud_type, SFVec3d& position,
```

```
            SFVec3d& size, SFFloat& density, SFFloat& thickness) const;
void    SetCloudProperty(SFInt cloud_type, const SFVec3d& position,
            const SFVec3d& size, SFFloat density, SFFloat thickness);
void    GetSunProperty(SFVec3d& position, SFColorRGB& color) const;
void    SetSunProperty(const SFVec3d&position, const SFColorRGB&color);
void    GetMoonProperty(SFVec3d& position, SFColorRGB& color) const;
void    SetMoonProperty(const SFVec3d&position, const SFColorRGB&color);
void    GetSkylightProperty(SFColorRGB& ambient,
            SFColorRGB& horizon_color, SFString& texture_name) const;
void    SetSkylightProperty(const SFColorRGB& ambient,
            const SFColorRGB& horizon_color, const SFString& texture_name);
```

In the function GetCloudProperty and SetCloudProperty, the value of the parameter cloud_type can be CLOUD_CIRROCUMULUS, CLOUD_CIRRUS, CLOUD_STRATUS and their combination. The parameter position specifies the center of clouds, represented by the geodetic spatial reference frame coordinates or the world frame coordinates according to the environment's reference frame setting. The parameter size describes the bounding box of clouds. The parameter density specifies the volume ratio of the clouds and their bounding box. The visibility inside clouds is given by the parameter thickness.

If the geodetic spatial reference frame is adopted, the position of the sun would be neglected in the function SetSunProperty since the sun's position can be calculated according to the position and time of the environment. So can the moon's position in the function SetMoonProperty.

4.3.3 vxIUI

As mentioned above, the class vxIUI is designed to be a façade to manage all user interface stuffs in the VCRS. The representation of the user interface of the VCRS is organized as a hierarchical structure, which is called the UI tree. The root of the UI tree represents a display space. The children nodes of this display space node represent a set of screens. The display space has a 2D coordinate frame, display space frame, where the display region of each screen is defined in this frame. The children nodes of a screen node are the window nodes representing windows on this screen. Each window could have one or more child windows, UI controls or viewports. It means the children of a window node can be window nodes, UI control nodes and viewports nodes. Fig. 4.2 illustrates the UI tree through an example. The UI tree of an application is built up from the configuration file in the startup stage of the rendering system (see subsection 4.3.1 "vxIRenderingSystem").

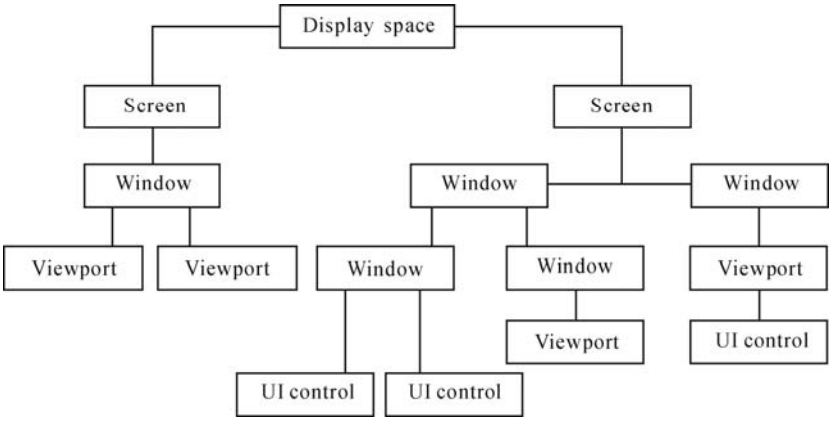

Fig. 4.2 An example of the UI tree

In most cases, a screen is in charge of one or more render servers. For a multiple tiled display system, its screen corresponds to an array of render servers, where each render server takes charge of a display tile. However, there is one exception. For a user application that requires both the Microsoft Windows's GUI and VCRS' UI in the same process on the same machine, the user application should create a virtual local screen, and place one or more of the VCRS' windows on the screen. Because the application and the render controller run in the same process, the virtual local screen is in charge of a local rendering engine rather than a render server. The following function adds a virtual local screen which is a window created by the operating system. The window is represented by a handle.

```
SFInt       AddVirtualLocalScreen(SFInt hwnd);
```

Each viewport owns a camera, which provides viewing parameters for rendering a scene. Moreover, a viewport corresponds to a scene graph or a subgraph. The correspondence can be either statically configured via the configuration file or dynamically configured in runtime.

In the following, some of the important public member functions of vxIUI are introduced:

```
SFInt       GetScreenCount() const;
vxIScreen   GetScreen(SFInt index);
SFInt       GetScreenIndex(const SFString& name);
SFInt       GetScreenIndex(SFID id);
```

These functions are used to get screen interfaces. Since any screen object must correspond to a physical display system, it can be created automatically by the VCRS from a configuration file. It is not necessary for developers to explicitly create screen objects in their applications. Therefore, the class vxIUI does not

provide any functions for screen object creation and deletion.

```
void        GetBoundingRect(SFVec2f& bottom_left,
                SFVec2f& top_right) const;
```

This function obtains the bounding rectangle of all screens in the display space frame.

```
SFInt       SendMessage(vxCMessage& msg );
SFInt       PostMessage(vxCMessage& msg);
```

The two functions send or post messages to screens, windows, viewports or UI controls. The function SendMessage works in a blocking fashion, and the other does so in a non-blocking fashion. The structure vxCMessage is declared as follows.

```
struct vxCMessage
{
    SFID        target,
    SFInt32     message;
    SFUInt16    wparam;
    SFInt32     lparam;
};
```

4.3.3.1 vxIScreen

The class vxIScreen is the interface to represent screens. The main public member functions are introduced as follows. For a screen, the origin of its screen coordinate frame is (0, 0). The maximum valid coordinates along the *x*-axis and *y*-axis are specified by the screen horizontal and vertical resolution respectively.

- *Basic*

```
SFBool            IsValid() const;
const SFString&   GetName() const;
SFID              GetID() const;
SFBool            IsVirtualLocalScreen() const;
void              GetRegion(SFVec2f& bottom_left,
                      SFVec2f& top_right) const;
void              GetSize(SFInt& width, SFInt& height) const;
void              SetSize(SFInt width, SFInt height) const;
```

These functions get the basic properties about the screen. The values of these properties are given in the configuration file. The function GetRegion obtains the

rectangle of the screen in the display space frame.

In most cases, the size of the screen is determined by the physical display device, so it cannot be set during the runtime by the applications. However, it is valid for the local virtual screen, since this kind of screen is in fact a window and its size can be changed freely.

- *Window*

```
SFInt       AddNewWindow(const SFString& name, SFUInt window_style,
                 const SFVec2i&left_bottom, const SFVec2i&
                 right_top, vxIWindow parent, vxIUIControl menu);
void        RemoveWindow(SFInt index);
vxIWindow   GetWindow (SFInt index) const;
SFInt       GetWindowCount() const;
SFInt       GetWindowIndex(const SFString& name) const;
SFInt       GetWindowIndex(SFID id) const;
SFInt       GetWindowIndex (SFInt x, SFInt y) const;
```

This group of functions is designed to create, remove and obtain a window. The function AddNewWindow creates a new window and adds it to the screen. The parameter left_bottom and right_top defines a rectangle region on the screen to place the window. The function GetWindowIndex(SFID id) translates the window identifier to its index on the screen. The last function obtains the index of the window where covers point (*x*, *y*) on the screen coordinate frame.

- *UI control*

```
VxIUIControl    CreateUIControl2D(SFInt control_type,
                    SFUInt control_style, const SFVec2i&
                    left_bottom, const SFVec2i& right_top,
                    vxIUIControl parent);
  vxIUIControl  CreateUIControl3D(SFInt control_type,
                    SFUInt control_style,const SFVec3d& center,
                    const SFVec3d& size, vxIUIControl parent);
  void          DestroyUIControl(vxIUIControl control);
  void          SetUIControlFont(SFInt size, SFInt type,
                    const SFColorRGB& color);
```

The two functions create various UI 2D/3D controls. The UI control can be the following types; button, static text, text box, combo box, list box, check box, group, picture, progress bar, menu, video playback, annotation, transform widget, etc. For 2D UI controls, their position and size is determined by the two corners on the screen coordinate frame. For 3D ones, their position and size is determined by their bounding boxes' center and size in the relevant scene graph's world coordinate frame.

Although the VCRS has a garbage clean, it is more efficient for the developer

explicitly to destroy unused UI controls when the UI controls are created and removed frequently. Therefore, the function DestroyUIControl is provided to forcefully destroy a UI control regardless of whether it has been referred somewhere else. The developers should guarantee any owner of the destroyed control to remove this control.

The function SetUIControlFont sets the font size, type and color for any UI control shown on this screen.

- *Message*

```
SFInt     AddMessageHandler(vxIScreenMessageHandler&Handler);
```

This function adds a message handler to the screen. Application developers should override the class vxIScreenMessageHandler to customize message processing. The class vxIScreenMessageHandler is declared as follows.

```
class vxIScreenMessageHandler
{
public:
     ~virtual vxIScreenMessageHandler();
     virtual vxIScreenMessageHandler* Clone()=0;
     virtual SFInt Process(vxIScreen& screen, vxCMessage& message);
};
```

The abstract function Clone should create an instance of the concrete class. Here the design pattern prototype is adopted. The virtual function Process should be overridden to process the input message.

4.3.3.2 vxIWindow

The vxIWindow is the interface of window objects. A window in the VCRS can be regarded as a container of child windows, viewports and UI controls.

- *Basic*

```
SFBool          IsValid() const;
const SFString& GetName() const;
SFID            GetID() const;
SFBool          IsVisible() const;
void            Show(SFBool to_show);
void            GetPosition(SFVec2i&left_bottom, SFFloat& z);
void            SetPosition(const SFVec2i& left_bottom, SFFloat z);
void            GetSize(SFVec2i& size) const;
void            SetSize(const SFVec2i& size);
```

The function Show shows or hides the window according to the Boolean parameter to_show. The function IsVisible queries the visibility of the window.

- *Viewport*

```
SFInt        AddNewViewport(const SFString&name,
                    const SFVec2i&left_bottom,
                    const SFVec2i&right_top);
void         RemoveViewport (SFInt index);
vxIViewport  GetViewport (SFInt index) const;
SFInt        GetViewportCount() const;
SFInt        GetViewportIndex(const SFString&name) const;
SFInt        GetViewportIndex(SFID id) const;
SFInt        GetViewportIndex (SFInt x, SFInt y) const;
```

This group of functions creates, removes and gets viewports in the window.

- *Child window*

```
SFInt      AddNewWindow(const SFString& name, SFUInt window_style,
                 const SFVec2i&left_bottom, const SFVec2i&
right_top, vxIUIControl menu);
void RemoveWindow(SFInt index);
vxIWindow  GetWindow (SFInt index) const;
SFInt      GetWindowCount() const;
SFInt      GetWindowIndex(const SFString& name) const;
SFInt      GetWindowIndex(SFID id) const;
SFInt      GetWindowIndex (SFInt x, SFInt y) const;
```

This group of functions manages child windows in the window.

- *UI control*

```
SFInt          AddUIControl(vxIUIControl control);
void           RemoveUIControl(SFInt index);
vxIUIControl   GetUIControl (SFInt index) const;
SFInt          GetUIControlCount() const;
SFInt          GetUIControlIndex(const SFString& name) const;
SFInt          GetUIControlIndex(SFID id) const;
```

This group of functions manages UI controls in the window.

- *Message*

```
SFInt    AddMessageHandler(vxIWindowMessageHandler& Handler);
```

This function adds a message handler to this window. The window message handler is similar to the screen message handler, except the input parameter passed to the member function Process is a window interface instead of a screen interface. The declaration of the class vxIWindowMessageHandler is shown in the following.

```
class vxIWindowMessageHandler
{
public:
      ~virtual vxIWindowMessageHandler();
      virtual vxIWindowMessageHandler* Clone()=0;
      virtual SFInt Process(vxIWindow& window, vxCMessage& message);
};
```

4.3.3.3 vxIViewport

The class vxIViewport is the interface of viewports. A viewport is a rectangle region for showing the rendering results of a scene graph or a part of a scene graph. Each viewport owns a camera and contains a reference to a scene graph node, from which the screen graph is traversed during the rendering. The main public member functions are introduced as follows.

- *Basic*

```
SFBool            IsValid() const;
const SFString&   GetName() const;
SFID              GetID() const;
SFBool            IsVisible() const;
void              GetRegion(SFVec2i&left_bottom,
                        SFVec2i& right_top) const;
```

If the parent window is visible, the function of IsVisible returns true.

- *Scene graph*

```
vxISceneGraph    GetSceneGraph() const;
vxISGNode        GetSceneGraphNode() const;
void             SetSceneGraph(vxISceneGraph scene_graph);
void             SetSceneGraphNode(vxISGNode scene_graph_node);
```

The functions get and set the root for traversing the scene model during the rendering. The root can be either a scene graph or a node of a scene graph.

- *Camera and view*

```
vxICamera        GetCamera();
```

This function returns the interface of this viewport's camera. Developers can directly call this interface's member function to manipulate the camera.

```
SFInt            SaveCurrentView(const SFString& view_name,
                     SFBool take_picture,
                     const SFVec2i& picture_size);
void             ReappearView(SFInt index);
void             RemoveView(SFInt index);
SFInt            GetViewCount() const;
SFID             GetViewID(SFInt index) const;
SFInt            GetViewIndex(const SFString& name) const;
SFInt            GetViewIndex(SFID id) const;
const SFString&  GetViewName(SFInt index) const;
```

The concept of view here is a set of viewing parameters, namely a snapshot of a camera's states. The function SaveCurrentView creates a view object which is a snapshot of the current camera states, and returns the view's index stored in this viewport. The function ReappearView updates the current camera states with the specified view object.

```
SFBool           StartRecordingViewSequence(const SFString&
                     sequence_name);
SFInt            StopRecordingViewSequence();
SFBool           StartPlayingViewSequence(SFInt index);
void             StopPlayingViewSequence();
SFBool           ExportViewSequenceVideo(SFInt index,
                     const SFString& video_name);
void             RemoveViewSequence(SFInt index);
SFInt            GetViewSequenceCount() const;
SFInt            GetViewSequenceIndex(const SFString& name) const;
SFInt            GetViewSequenceName(SFInt index) const;
```

The above functions manage the view sequence objects, each of which is an array of the view objects. The function StartRecordingViewSequence requests this viewport object to record a view sequence, and the function StopRecordingViewSequence stops the recording. Invoking the function StartPlayingViewSequence will cause the viewport to continuously update its camera with the views in the view sequence, and the function StopPlayingViewSequence stops the updating process. The sequence of images, each of which is a snapshot of the view in the specified view sequence, is recorded and packed into a video file by the function ExportViewSequenceVideo.

- *Walkthrough*

```
void     StartWalkthrough(SFInt motion_type, const SFString&
             config_filepath);
void     StopWalkthrough();
```

When the function StartWalkthrough is invoked, the camera of this viewport will be in a certain motion type so as to simulate piloting a plane, steering a ship, driving a car, or just walking around. The first parameter motion_type can be one of the following values, PILOTING_PLANE, PILOTING_HELICOPTER, STEERING_SHIP, PILOTING_SUBMARINE, DRIVING_CAR, DRIVING_MOTORCYCLE, HUMAN_CYCLING, HUMAN_WALKING. The second parameter config_filepath specified the motion configuration file, in which the parameters for such simulation are defined. The camera motion would be stopped once the function StopWalkthrough is invoked. The following functions are designed to set walkthrough parameters.

```
void    SetAutoWalkthrough (const MFVec3d& destination,
            const MFVec3d& travel_time, const MFVec3d& staying_time);
void    SetAutoWalkthrough (const MFInt& index_of_destination_view,
            const MFVec3d& travel_time, const MFVec3d& staying_time);
void    SetAutoWalkthrough(vxISGNode object, SFInt tracking_mode,
            const SFVec3f& camera_viewpoint, const SFVec3f&
            camera_target, const SFVec3f& camera_up_dir,
            const SFVec3f& camera_surrounding_dir);
```

The function SetAutoWalkthrough specifies in which motion the camera would be if no user input were provided. This function has three versions. The first version indicates one or more destinations that the camera should arrive at successively, and sets the travel time and staying time for every destination. The second version is similar to the first one, except that the recorded views are used as the destinations. The third one makes the camera always follow an object or surround it. The specified camera's viewpoint, target, up direction and surrounding direction all are defined in the local coordinate frame of the followed object.

```
void    SetWalkthroughObstacleDetecting(SFBool enable_flag);
void    SetWalkthroughAtFixedHeightAboveSea(SFBool enable_flag,
            SFFloat height);
void    SetWalkthroughAtFixedHeightAboveGround(SFBool enable_flag,
            SFFloat height);
```

The above three functions enable or disable the walkthrough options. The first one sets the obstacle detecting option. It means the camera will not pass through any obstacle. The function SetWalkthroughAtFixedHeightAboveSea constrains the motion of the camera at the specified height above sea level, or releases the

constraint. The third function places the camera always at the given height above ground level. Once the function is called, the camera motion path is wavy if the ground is hilly.

```
void    SetWalkthroughAcceleration(const SFVec3d& a);
void    SetYawAcceleration(SFDouble aa);
void    SetPitchAcceleration(SFDouble aa);
void    SetRollAcceleration(SFDouble aa);
```

The function SetWalkthroughAcceleration specifies the acceleration of the camera. The other functions give the angular acceleration in three directions. Pitching the camera makes its look in an up and down direction, just like rotating the camera around its *X*-axis. A yaw is rotation around the *Y*-axis and roll is rotation around the *Z*-axis (the look direction).

- *Viewport coordinate*

The coordinate frame of the viewport is referred to as the viewport coordinate frame.

```
void    IsInViewport(const SFVec2i& point) const;
```

The above function checks whether a point is in this viewport. The coordinates of this point are in the screen coordinate frame.

```
void    ScreenToViewport (const SFVec2i& screen_pt,
            SFVec2i& viewport_pt) const;
void    ViewportToScreen (const SFVec2i& viewport_pt,
            SFVec2i& screen_pt) const;
```

The above two functions transform the point coordinates either from the screen coordinate frame to the viewport coordinate frame or from the viewport coordinate frame to the screen coordinate frame.

```
void    ViewportToWorld(const SFVec2i& viewport_pt,
            SFVec3d& world_ray_start_pt, sFVec3d&
            world_ray_direction) const;
SFBool  ViewportToWorld(const SFVec2i& viewport_pt,
            SFVec3d& world_pt) const
SFBool  ViewportToWorld(const SFVec2i& viewport_pt,
            SFVec3d& world_pt, const SFVec3d& ref_plane_pt,
            const SFVec3d& ref_plane_normal) const;
void    WorldToViewport(const SFVec3d& point,
            SFVec2i& viewport_pt) const;
```

The above functions provide the coordinate transform from the viewport coordinate frame to the world coordinate frame. The function ViewportToWorld has three versions.

The first version calculates a ray which starts from the camera center and passes through the point specified by the input parameter viewport_pt. Since the viewport coordinate frame is actually built on the camera's projection plane if we view it in 3D space, this 2D point implicitly and uniquely specifies one 3D point on this projection plane.

The second version calculates the first intersection point between the aforementioned ray and the scene relevant to this viewport. It should be noted that the coordinates of the intersection point are in the world coordinate frame of the scene graph, not in the coordinate frame of the root node of the sub graph.

The third version calculates the intersection point between the ray and a reference plane.

- *Message*

```
SFInt   AddMessageHandler(vxIViewportMessageHandler& Handler);
```

The viewport message handler is similar to the screen message handler or the window message handler, except that the input parameter passed to the member function Process(vxIViewport& viewport, vxCMessage& message) is a viewport interface.

4.3.3.4 vxICamera

The class vxICamera is the interface of camera objects.

- *Looking at and projecting settings*

```
void    SetLookingInfo(const SFVec3d& viewpoint, const SFVec3d&
            target, const SFVec3d& up_direction);
void    SetPerspectiveFrustum(SFDouble fovy, SFDouble aspect,
            SFDouble near, SFDouble far);
void    SetFrustum (SFInt projection_type, SFDouble left,
            SFDouble right, SFDouble bottom, SFDouble top,
            SFDouble near, SFDouble far);
void    SetLookingMatrix(const SFMattrix44d& m);
void    SetProjectingMatrix(const SFMatrix44d& m);
void    GetLookingInfo(SFVec3d& viewpoint, SFVec3d& target,
            SFVec3d& up_direction) const;
void    GetPerspectiveFrustum(SFDouble& fovy SFDouble& aspect,
            SFDouble& near, SFDouble& far) const;
```

```
void    GetFrustum(SFInt& projection_type, SFDouble& left,
            SFDouble& right, SFDouble& bottom, SFDouble& top,
            SFDouble& near, SFDouble& far) const ;
void    GetLookingMatrix(SFMattrix44d& m);
void    GetProjectingMatrix(SFMatrix44d& m);
```

This group of functions sets and obtains the camera coordinate frame and perspective or orthogonal projection settings. The function SetLookingInfo and SetPerspectiveFrustum are similar to the function gluLookAt and gluPerspective in OpenGL respectively. The former function sets the camera coordinate frame and the rear one sets the perspective projection parameters.

The function SetFrustum sets the viewing frustum for either the perspective or orthogonal projecting case. The first parameter projection_type specifies the projection type.

The viewing matrix, which makes the coordinate transform from the world coordinate frame to the camera coordinate frame, can be directly set by the function SetLookingMatrix. This function is equivalent to the function SetLookingInfo. The projecting matrix can be set by the function SetProjectingMatrix.

- *Movement and rotation*

```
void    Move (const SFVec3d& movement);
void    MoveTowards(const SFVec3d& destination,
            SFDouble moving_distance_ratio);
void    MoveTowardsBox(const SFBox& box);
void    Pan (const SFVec2i& current_pos,
            const SFVec2i& destination_pos);
void    Rotate (SFFloat radian, const SFVec3d& axis_position,
            const SFVec3d& axis_direction);
void    Rotate(const SFQuatertion& q);
void    RotateOnGravityAxis (SFDouble radian);
```

This group of functions makes the camera coordinate frame move one step, or rotate once. The function Move is designed to move the camera, and the moving distance and direction are specified by the input vector movement. The coordinates of the vector movement are in the world coordinate frame.

The function MoveTowards moves the camera towards the position specified by the parameter destination, and the ratio of the moving distance to the total distance between the current position of the camera and the destination is given by the parameter moving_distance_ratio. The parameter moving_distance_ratio can be a negative value, which means the camera moves far away from the destination.

The function MoveTowardsBox moves the camera towards the center of the given box until the entire box can just be seen in the current viewport.

The function Pan moves the camera on its projecting plane so that the projection of a certain 3D point moves from the position current_pos to destination_pos. The

coordinate of the position is in the viewport coordinate frame.

There are two versions of the function Rotate, which rotates the camera, more precisely the camera coordinate frame. The function of the first version rotates the camera on the given axis that passes through the position axis_position and is aligned with the direction axis_direction. The rotation angle is specified by the parameter radian. The function of the second version just applies a rotation represented by the input quaternion q to the camera.

The function RotateOnGravityAxis rotates the camera on the gravity axis which passes through the origin of the camera coordinate frame and is aligned with the gravity direction.

4.3.3.5 vxIUIControl

The class vxIUIControl is the interface of various user interaction elements. Currently, the VCRS defines the UI control types including button, static text, text box, combo box, list box, check box, radio group, picture, process bar, menu, annotation, video playback and transform widget. The creation of any UI controls is performed by screen objects, as is introduced in subsection 4.3.3.1 “vxIScreen”. There are two kinds of creation, 2D or 3D. The layout of UI controls in a window or a viewport can be specified in a UI resource file (see subsection 4.3.1 “vxIRenderingSystem”).

The main public functions are introduced in the following.

- *Basic*

```
SFBool            IsValid() const;
SFInt             GetUIControlType() const;
const SFString&   GetName() const;
SFID              GetID() const;
SFBool            IsVisible() const;
void              Show(SFBool to_show);
void              GetPosition(SFVec2i& left_bottom, SFFloat& z);
void              SetPosition(const SFVec2i& left_bottom, SFFloat z);
void              GetSize(SFVec2i& size) const;
void              SetSize(const SFVec2i& size);
```

The function GetUIControlType returns the type of the concrete UI control represented by the interface.

- *Message*

```
SFInt       AddMessageHandler(vxIUIControlMessageHandler& Handler);
```

The class vxIUIControlMessageHandler is similar to the prior message handler

classes, except the member function Process is declared as follows.

```
virtual vxIUIControlMessageHandler::SFInt Process(vxIUIControl&
control, vxCMessage& message);
```

- *Properties*

For each kind of UI controls, there are a group of functions to get and set its properties. These properties are similar to those of Microsoft Window GUI controls. Therefore, we did not introduce them in detail.

4.3.4 The Basic Example

In this section we will show how to exploit the VCRS to develop a basic walkthrough application. In this example there are two screens. One is a virtual local screen, which in fact is a window in the operating system of Microsoft Windows. The other is a large-scale tiled display. Both screens synchronously show the same 3D graphic contents. Users can manipulate the camera during the walkthrough via a mouse.

```
// CMyWnd.h
class CMyWnd : public CWnd
{
protected:
    vxIRenderingSystem   m_system;
    vxIScreen            m_screen0, m_screen1;
    vxIViewport          m_viewport0, m_viewport1;

    DECLARE_MESSAGE_MAP()
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnSize(UINT nType, int cx, int cy);
    afx_msg void OnPaint();
    afx_msg void OnFileOpen();
...
};

// CMyWnd.cpp
BEGIN_MESSAGE_MAP(CMyWnd, CWnd)
    ON_WM_CREATE()
    ON_COMMAND(ID_FILE_OPEN, OnFileOpen)
    ON_WM_SIZE()
    ON_WM_PAINT()
    ON_WM_MOUSEMOVE()
```

```
END_MESSAGE_MAP()

int CMyWindow::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    vxCRenderingSystemFactory factory;
    m_system = factory.Create();

    if(m_system == NULL)
        return false;
    if(!m_system.StartUp("d:\\vxconfigure.cfg"))
         return false;

HWND hwnd = CreateWindow(...);
...

vxIUI   ui = m_system.GetUI();
SFInt   si0 = ui.AddVirtualLocalScreen(hwnd);
if(si0 < 0) return false;
m_screen0 = ui.GetScreen(si0);
SFInt iw = m_screen0.AddNewWindow("", WS_FULLSCREEN,...);
SFInt iv = m_screen0.GetWindow(iw).AddNewViewport("",...);
m_viewport0 = m_screen0.GetWindow(iw).GetViewport(iv);

SFInt   si1 = ui.GetScreenIndex("large_scale_tiled_display");
If(si1 < 0) return false;
m_screen1 = ui.GetScreen(si1);
iw = m_screen1.AddNewWindow("" , WS_FULLSCREEN,...);
iv = m_screen1.GetWindow(iw).AddNewViewport("",...);
m_viewport1 = m_screen1.GetWindow(iw).GetViewport(iv);
}

void CMyWindow::OnFileOpen ()
{
    CFileDialog dlg(TRUE, "xw", NULL, 0, "*.xw ");
    if(dlg.DoModal() == IDOK){
         if(!m_system.GetModel().OpenSceneModel((LPCSTR)dlg.
               GetPathName()) return false;
         }
    }
    m_viewport0.SetSceneGraph(model.GetSceneGraph(0));
    m_viewport1.SetSceneGraph(model.GetSceneGraph(0));
}

void CMyWindow::OnSize(UINT nType, int cx, int cy)
{
    m_screen0.SetSize(cx, cy);
```

```
}

void CMyWindow::OnPaint()
{
    m_system.DoRendering();
    m_system.SwapBuffer();
}

void CMyWindow::OnMouseMove(UINT nFlags, CPoint point)
{
    SFVec3d viewport, target, up_direction;
    Set the vectors viewpoint, target, up_direction according to
        the mouse movement and flag.
    vxICamera camera0 = m_viewport0.GetCamera();
    vxICamera camera1 = m_viewport1.GetCamera();
    camera0.SetLookingInfo(viewpoint, target, up_direction);
    camera1.SetLookingInfo(viewpoint, target, up_direction);
    OnPaint();
}
```

4.4 Server Manager

4.4.1 Functionality

The server manager is represented by the class CServerManager, the functionality of which includes calling remote services and managing all render servers to keep them working synchronously. The rendering system interface provides application developers with various local services to ease the development efforts. In view of these developers, the provider of a local service is usually a scene model object or a user interface object. However, these local services should be executed remotely by the relevant render servers. Therefore, to realize the remote service calling, the server manager must translate the local service requests into the corresponding remote service requests. It means that the relevant render servers which provide the remote services are found first, and then the corresponding service requests are sent to them.

4.4.2 Structure

The structure and data flow of the server manager are illustrated in Fig. 4.3. It can be considered as three layers.

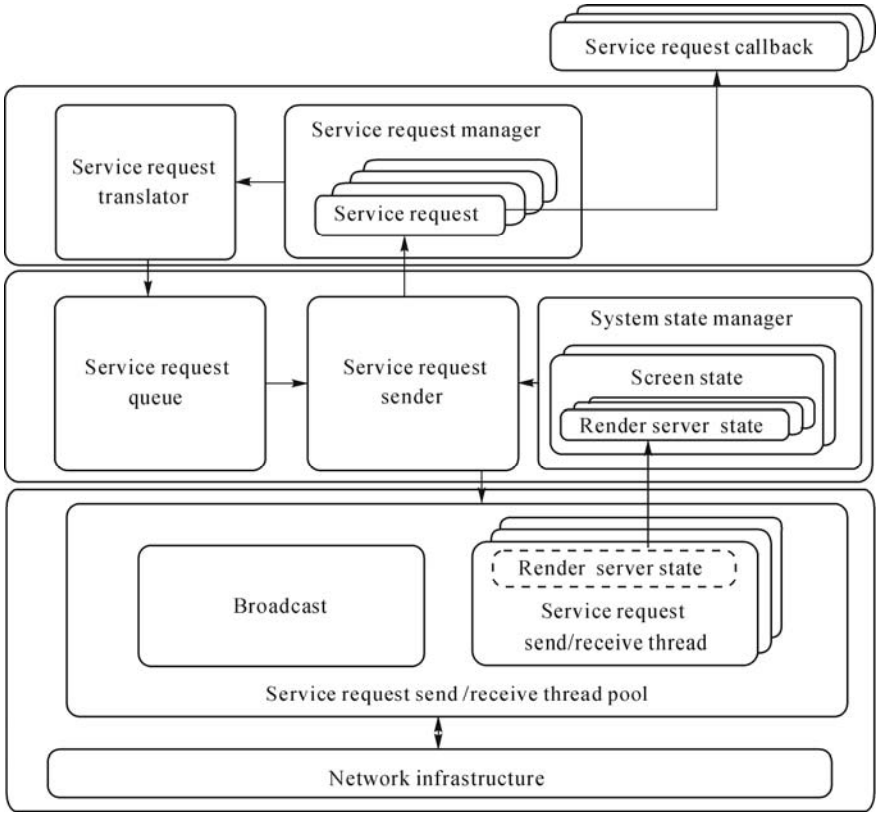

Fig. 4.3 The structure and data flow of the server manager

4.4.2.1 Layer one

The first layer is designed as the input and output interface of the server manager. In this layer, a variety of service requests are defined. Any service request is represented by a subclass of the abstract class IServiceRequest. A service request object encapsulates the service type and the service provider together with all other necessary parameters. In addition, it also provides a field to hold the outcomes of the service, which will be passed into a service callback function after this service request has been performed remotely. Moreover, either local service requests or remote ones can be represented by the instances of the subclasses of IServiceRequest.

All service request instances are created by the module service request manager, the instance of the class CServiceRequestManager. The service request manager takes control of the lifetime of any service request instance. In addition, the manager provides a few functions to take in local service requests.

After a local service request is accepted by the service request manager, it is then passed to another module service request translator, the instance of the class CServiceRequestTranslator. The main function of the class CServiceRequestTranslator is to translate the service requests from a local version to a remote version. The nature of such translation is to find the appropriate render servers which should respond to the requested service. The addresses of these render servers are added to the service request object.

4.4.2.2 Layer two

The second layer is designed to deal with service request queuing, monitor the states of all render servers, and give service outcomes back to service requestors. Moreover, it handles the synchronization of the rendering servers.

The states of all render servers are kept in the module system state manager. The system state manager manages a group of the screen state objects, each of which depicts the state on a screen in the system. Since, in the VCRS, the display contents on each screen are rendered by one or more render servers, the screen state can also be regarded as the state set for the render servers relevant to the screen. Therefore, each screen state object manages a set of the render server states.

For the service requests which are not required be responded to right now, they are placed in the service request queue, waiting for the service request sender to fetch and send them. This queue is a priority queue, and the service request sender checks each request's priority to determine which is to be sent first. For any service request which needs be responded to immediately, it will be immediately sent to the corresponding render server by the service request sender.

To obtain feedback of the fulfillment of a local service, the feedbacks reported from all the relevant render servers should be collected and summarized by the service request sender. The service request sender sets the summary result in the corresponding service request, and invokes the callback function to notify the requestor the service has already been fulfilled and the outcome is ready.

In many cases, only the requestor of the local service knows the outcome of fulfilling the local service. It will know what to do in the next step. In other words, if a service request has not been remotely fulfilled by all render servers yet, any codes following the local service request will not be executed. Such kind of service request is named blocking service request.

4.4.2.3 Layer three

This layer is built based on the acceptor-connector and proactor framework of the ADAPTIVE Communication Environment (ACE), which is a network

application development toolkit, and its online address is http://www.cs.wustl.edu/~schmidt/ACE.html. Each render server corresponds to a service request send/receive thread, and all of them are put in a thread pool. Each send/receive thread has a reference to the relevant render server state object, which is kept in the system state manager. The render server state is used to monitor the render server as well as to record the outcomes of fulfilling service requests.

4.4.3 CServerManager

The main public member functions of the class CServerManager are listed as follows.

```
SFInt     StartUp(CSystemConfig& system_configuration);
void      ShutDown();
```

The two functions start up and shut down the server manager respectively. In the function StartUpServerManager, the specified system configuration is given to configure either the server manager or the remote render servers. During the process of the start up, the network connections between the render controller and all render servers are established.

```
CServiceRequestManager* GetServiceRequestManager();
```

This function retrieves the pointer to the aggregated service request manager.

4.4.4 CServiceRequestManager

The main public member functions of the class CServiceRequestManager are shown as fallows.

```
IServiceRequest*        Give(SFInt type);
SFInt     SendServiceRequest (IServiceRequest * pRequest);
SFInt     SendServiceRequest (vxIScreen& screen,
               IServiceRequest * pRequest);
SFInt     SendServiceRequest (vxIWindow& window,
               IServiceRequest * pRequest);
SFInt     SendServiceRequest (vxIViewport& viewport,
               IServiceRequest * pRequest);
```

The function Give creates a service request instance according to the specified service request type. The manager takes control of the lifetime of all service

request instances. It means these instances must not be released outside this manager. Inside the manager, there is a memory pool to keep all created service request instances.

There are four versions of the function SendServiceRequest for different service providers. The first version does not explicitly specify the service provider. It means the service provider is the object of the class vxIRenderingSystem or vxIModel. Because either of them has only one instance in the whole system, it is not necessary to specify it in the function SendServiceRequest. The other three versions specify a screen, a window and a viewport as the service providers, respectively.

The composition of the class CServiceRequestManager is shown as follows:

```
class CServiceRequestManager
{
     CServiceRequestTranslator     m_Translator;
     CServiceRequestSender         m_Sender;
     CSystemStateManager           m_StateMgr;
     CServiceRequestQueue          m_Queue;
     CServiceRequestSRThreadPool   m_ThreadPool;
}
```

The pseudo codes to implement the first version of the function SendServiceRequest are listed in the following:

```
SFInt CServiceRequestManager::SendServiceRequest(IServiceRequest *
     pRequest)
{
     pRequest->Serialize();
     m_Translator.Translate(pRequest);
     If(this request should be sent immediately)
          rtn = m_Sender.SendImmediate(pRequest);
     else
          rtn = m_ServiceRequestQueue.PushBack(pRequest);
     return rtn ;
}
```

There are a variety of service requests, each of which is represented by a subclass of the class IServiceRequest. These classes are introduced later.

The following function WaitForFeedback blocks the thread to wait for the feedback of the specified service request. The second parameter indicates whether the resources related to this request should be released automatically after the feedback is returned. The function Release is provided to make the manual resource release.

```
SFInt CServiceRequestManager::WaitForFeedback(IServiceRequest *
```

```
    pRequest, SFBool release_resource = true)
{
    return m_Sender.WaitForFeedback(pRequest, release_resouce);
}
void CServiceRequestManager::Release(IServiceRequest * pRequest);
```

4.4.5 CServiceRequestTranslator

The main functionality of the class CServiceRequestTranslator is to find the render servers which would provide the remote services according to the specified local service providers and fill the network addresses in the input service request instances. It declares the four translation functions as follows:

```
SFInt   Translate(IServiceRequest * pSvcRequest);
SFInt   Translate(vxIScreen& screen, IServiceRequest * pSvcRequest);
SFInt   Translate(vxIWindow& window, IServiceRequest * pSvcRequest);
SFInt   Translate(vxIViewport& viewport, IServiceRequest * pSvcRequest);
```

4.4.6 CServiceRequestSender

The class CServiceRequestSender sends a service request either in immediate mode or in asynchronous mode.

The function SendImmediate sends the specified service request immediately. For a blocking service request, the current thread is blocked and waits for all render servers to give their feedbacks. The service outcome is filled in the input server request object. If a request needs feedback, it will be added to the system state manager so that its feedback can be queried or output through callback. The pseudo codes are listed as follows:

```
void CServiceRequestSender::SendImmediate(IServiceRequest* pRequest)
{
    Send the service request to corresponding render servers right now;
    If(the request is a blocking service request)
    {
        Do{
            Collect all render servers' feedbacks from the render
            server states kept in system state manager;
        Until(all render servers have given their feedbacks or
        the timeout is expired);
        Summary these feedbacks to obtain the outcome of this
```

```
            service request;
            Fill the outcome in the service request object;
        }Else if(the service request needs a feedback)
        {
            Add the service request information into the system state
            manager;
        }
    }
```

Sending a service request in asynchronous mode is a little bit more complicated than doing it in immediate mode. The service requests which will be sent in asynchronous mode are all placed in the service request queue. The service request sender periodically picks the service requests from the queue and sends them. The function Perform is just designed to do this job. This function is called periodically by the framework. It performs two tasks. One is to check the service request queue and send it to target render servers through the relevant service request send/receive threads respectively. The other is to summarize the feedbacks of a number of remote service providers so as to form the overall feedback of the local service. The service requestor gets the overall feedback through callback or query. The pseudo codes are listed as follows.

```
void CServiceRequestSender::Perform()
{
    Pop up the service request which has the top priority from the
    service request queue;
    Get the render server list from the service request;
    For each render server on the list
    {
        Find the service request send/receive thread corresponding
        to the render server;
        Send the service request through the thread;
    }

    If (the service request needs a feedback)
        Add the service request information into the system state
        manager

    For each service request that needs feedback recorded by the
    system state manager
    {
        Collect all render servers' feedbacks of this service request
        from the render server states kept in the system state manager;
        If(all render servers have given their feedbacks)
        {
            Summarize these feedbacks to form the overall feedback;
```

```
            Record the overall feedback in the service request;
            If(the service request needs a callback)
            {
                Invoke the relevant callback function;
                Release the resources related to this service request;
            }
        }
    }
}
```

The function WaitForFeedback, shown in the following, blocks the thread to periodically query the overall feedback of the specified service request:

```
SFInt CServiceRequestSender::WaitForFeedback(IServiceRequest *
pRequest, SFBool release_resources = true)
{
   While(the overall feedback of this request has not been obtained yet)
       Perform();
   SFInt feedback = the overall feedback of this request;
   If(release_resouces)
       Release the resources related to this service request;
   Return feedback;
}
```

4.4.7 CSystemStateManager, CScreenState and CRenderServerState

The class CSystemStateManager is regarded as the aggregation of the class CScreenState, which is further regarded as the aggregation of the class CRenderServerState.

```
class CSystemStateManager
{
    TMF<CServiceRequestFeedback> m_feedbacks;
    CServiceRequestManager   m_service_request_manager;
    TMF<CScreenState> m_screen_states;
    // The properties depicting the common properties
    // of all render servers are declared in the following.
    …
};

struct CServiceRequestFeedback
{
    SFInt  service_request_id;
    SFInt  feedback;
```

```
    void*  buffer;
};

class CScreenState
{
    TMF<CRenderServerState>        m_render_server_states;
    TMF<CServiceRequestFeedback>   m_feedbacks;
    // The properties depicting the working states of a screen,
    // or say the common properties of the group of
    //the corresponding render servers are declared in the following.
...
};

class CRenderServerState
{
    TMF<CServiceRequestFeedback> m_feedbacks;
    // The properties depicting the working states
    // of a render server are declared in the following.
    ...
};
```

The array of CRenderServerState::m_feedbacks is used to track the feedbacks of service requests from a render server. The array CScreenState::m_feedbacks records the summarized feedbacks of several render servers corresponding to a screen. The array CSystemStateManage::m_feedbacks records the summarized feedbacks of all render server feedbacks. The following pseudo codes can unveil the relations between these arrays:

```
void CSystemStateManager::AddFeedback(IServiceRequest* pRequest)
{
    CServiceRequestFeedback feedback;
    feedback.service_request_id = pRequest->GetID();
    m_feedbacks.Add(feedback);
    int index;
    If((index=pRequest->GetScreenIndex())!= -1) // If the
    // provider is a screen
        m_screen_states[ index ].AddFeedback(feedback);
    Else
        For each element in m_screen_states, element.AddFeedback
        (feedback);
}

void CScreenState::AddFeedback(IServiceRequest* pRequest,
    const CServiceRequestFeedback& feedback)
{
    Obtain the render server list from the request;
    For each render server on the list
```

```
        Find the corresponding feedback m_feedbacks[index];
        m_feedbacks[index].Add(feedback);
}
```

4.4.8 CServiceRequestSRThreadPool

The class CServiceRequestSRThreadPool is designed based on the ACE Acceptor-Connector and ACE Proactor framework. It manages a set of threads for service request sending and receiving. The class is declared as follows:

```
class CServiceRequestSRThreadPool:
ACE_Asynch_Acceptor< CServiceRequestSRHandler >
{
public :
    bool StartListening(unsigned int port);
    void StopListening();
    CServiceRequestSRHandler * GetHandler(ACE_INET_Addr addr);
    void Broadcast(IServiceRequest * pRequest);
protected:
    virtual CServiceRequestSRHandler* make_handler();
    static ACE_THR_FUNC_RETURN EventLoopProc(void *arg);
}
```

The function StartListening will start a thread to listen to the specified port. Whenever a render server connects to this render controller, the pool will create a new thread with a new instance of the class CServiceRequestSRHandler to handle message sending and receiving through this connection, which is done by the function make_handler.

The function EventLoopProc is the message processing function of the listening thread, and it processes all events generated by the ACE framework.

The class CServiceRequestSRHandler inherits the class ACE_Service_Handler, which provides the message sending and receiving framework. Each instance takes charge of communicating with a render server, and it owns a reference to an instance of the class CRenderServerState, which records the state of the connected render server. The main part of the declaration of the class CServiceRequestSRHandler is listed as follows. Since it is quite an ordinary way to inherit the class ACE_Service_Handler, we are not going to give an explanation any more. For readers who have an interest in it, please make a reference to the website http://www.cs.wustl.edu/~schmidt/ACE.html, the online documents about the ACE.

```
class CServiceRequestSRHandler: public ACE_Service_Handler
{
public :
    virtual void open (ACE_HANDLE h, ACE_Message_Block&);
    virtual void addresses(const ACE_INET_Addr &remote_address,
                const ACE_INET_Addr &local_address);
```

```
    virtual void handle_read_stream(const ACE_Asynch_Read_Stream::
                 Result &result);
    virtual void handle_write_stream(const ACE_Asynch_Write_Stream::
                 Result &result);
    ...

private:
    CServiceRequestSRThreadPool   * m_ Pool;
    CRenderServerState            * m_pRenderServerState;
    ACE_Asynch_Read_Stream          m_Reader;
    ACE_Asynch_Write_Stream         m_Writer;
    ...
};
```

4.4.9 IServiceRequest and Subclasses

The class IServiceRequest is the base class of all service request classes. The main public member functions are listed as follows. All abstract functions should be overridden by the concrete classes.

```
virtual SFInt   GetType() const = 0;
SFInt           GetPriority() const;
SFInt           GetSendingType() const;
SFInt           GetFeedbackType() const;
virtual void    Serialize() = 0;
virtual SFBool  Unserialize() =0 ;
void            GetRenderServerIndexes(TMF<SFInt>& indexes) const;
SFInt           GetScreenIndex() const;
void            SetCallback(CServiceRequestCallback* callback);
```

The function GetType returns the type of concrete service request. If a request should be sent immediately, the function IsImmediate returns value true. Otherwise, it returns value false. If a request is a blocking request, the function IsBlocking returns value true.

The request sending type can be retrieved by the function GetSendingType, which returns one of the following values.

SENDING_IMMEDIATE: The request is sent immediately.

SENDING_IMMEDIATE_AND_WAIT: The request is sent immediately and the requestor's thread is blocked until all of the remote service providers give the feedback when they have finished their work. It is just for a blocking service request.

SENDING_ASYNCHRONOUS: The request is sent asynchronously.

The function GetFeedbackType returns one of the following values:

FEEDBACK_NONE: no feedback is needed;

FEEDBACK_QUERY: The feedback should be got by querying;

FEEDBACK_CALLBACK: The feedback will be passed into a callback.

For a blocking service request, or namely a request with the sending type value equal to SENDING_IMMEDIATE_AND_WAIT, the only valid value of the feedback type is neglected automatically.

The function Serialize serializes the parameters of this request into an internal buffer, and the function Unserialize parses the parameters from the buffer. It is the inverse operation of the serialization.

The function GetRenderServerIndexes returns the indexes of the corresponding render servers which will receive this request. The index of the screen, the display contents of which are rendered by these render servers, is obtained by the function GetScreenIndex.

The function SetCallback sets a pointer to an instance of the class CServiceRequestCallback, which represents a callback function. The class CServiceRequestCallback is just like a functor. If a callback is set to an immediate request, it is just ignored.

- *Subclasses*

The main subclasses of IServiceRequest are listed in Table 4.5. In this table, the default sending type and feedback type are also listed. These default values are initialized by the constructors of these subclasses respectively.

Table 4.5 The subclasses of the class IServiceRequest

Server request	Sending type	Feedback type
CSvcRqtSystemStartup	IMMEDIATE	QUERY
CSvcRqtSystemShutdown	ASYNCHRONOUS	CALLBACK
CSvcRqtSetRenderingOption	ASYNCHRONOUS	CALLBACK
CSvcRqtDoRendering	IMMEDIATE	CALLBACK
CSvcRqtSwapBuffer	IMMEDIATE	NONE
CSvcRqtCleanGarbage	ASYNCHRONOUS	CALLBACK
CSvcRqtCreateSceneModel	ASYNCHRONOUS	CALLBACK
CSvcRqtOpenSceneModel	ASYNCHRONOUS	CALLBACK
CSvcRqtCloseSceneModel	ASYNCHRONOUS	CALLBACK
CSvcRqtBeginTransaction	ASYNCHRONOUS	CALLBACK
CSvcRqtCreateFeature	ASYNCHRONOUS	CALLBACK
CSvcRqtImportScene	ASYNCHRONOUS	CALLBACK
CSvcRqtCreateSceneGraph	ASYNCHRONOUS	CALLBACK
CSvcRqtRemoveSceneGraph	ASYNCHRONOUS	CALLBACK
CSvcRqtBuildSpatialIndex	ASYNCHRONOUS	CALLBACK
CSvcRqtUpdateFeature	ASYNCHRONOUS	CALLBACK
CSvcRqtCommitTransaction	ASYNCHRONOUS	CALLBACK
CSvcRqtRollbackTransaction	ASYNCHRONOUS	CALLBACK
	IMMEDIATE	CALLBACK
CSvcRqtUpdateShadowRegion	ASYNCHRONOUS	CALLBACK
CSvcRqtCreateUI	ASYNCHRONOUS	CALLBACK
CSvcRqtUpdateUI	ASYNCHRONOUS	CALLBACK
CSvcRqtExportVideo	ASYNCHRONOUS	CALLBACK
CSvcRqtExportPicture	ASYNCHRONOUS	CALLBACK
CSvcRqtTransferFile	ASYNCHRONOUS	CALLBACK

If the service requests of class CSvcRqtCreateFeature, CSvcRqtUpdateFeature, CSvcRqtCreateSceneGraph and CSvcRqtRemoveSceneGraph are sent between the service requests of class CSvcRqtBeginTransaction and CSvcRqtCommitTransaction, the scene model in each render server is influenced in either the external memory or host memory. If they are not sent in a transaction, they only influence the scene model in the host memory.

4.5 Implementation of Rendering System Interface

In this section, we briefly introduce the principles for implementing the rendering system interface based on the VRE and the server manager. Moreover, we will show a few examples to give the reader a clear comprehension of the implementation.

4.5.1 Implementation Principles

- *Principle 1*

Local operation and remote service should be done in parallel. The relevant service requests should be sent before doing the local operations. When the local operations are finished, the thread is blocked to await the feedback. Another way is to use the callback fashion to obtain the feedback.

- *Principle 2*

Table 4.6 shows the correspondence between the interface classes and the service request. This table indicates which service requests are involved in an interface class. Some readers may find that various feature-related interface classes such as vxISGNode, vxILight and vxIShape merely correspond to one service request CSvcRqtUpdateFeature. It means that any changes taking place in these classes' instances will cause the update request to be sent. The update request will copy all contents of the local feature to update the remote counterparts. So does the class CSvcRqtUpdateUI.

- *Principle 3*

Either a temporal feature or persistent feature can be created through the same service request represented by the class CSvcRqtCreateFeature. If the request is sent just following a request to begin a transaction, it will be considered as a request to create a persistent feature by render servers. Otherwise, it is for temporal feature creation.

- *Principle 4*

The state changes of various user interface elements are performed through sending the requests of the class CSvcRqtUpdateUI. One special case is the viewport. Since the camera information is packed into the rendering request, which is represented by CSvcRqtDoRendering, and this rendering request is sent per frame, it is not necessary to pack the camera information in the viewport updating request.

Table 4.6 The correspondence between the interface classes and the service requests

Interface classes	Service requests
vxIRenderingSystem	CSvcRqtSystemStartup
	CSvcRqtSystemShutdown
	CSvcRqtSetRenderingOption
	CSvcRqtDoRendering
	CSvcRqtSwapBuffer
	CSvcRqtCleanGarbage
vxIModel	CSvcRqtCreateSceneModel
	CSvcRqtOpenSceneModel
	CSvcRqtCloseSceneModel
	CSvcRqtCreateSceneGraph
	CSvcRqtRemoveSceneGraph
	CSvcRqtCreateFeature
	CSvcRqtBeginTransaction
	CSvcRqtCommitTransaction
	CSvcRqtRollbackTransaction
	CSvcRqtImportScene
vxISceneGraph	CSvcRqtCreateTempFeature
	CSvcRqtUpdateFeature
	CSvcRqtUpdateShadowRegion
	CSvcRqtBuildSpatialIndex
vxISGNode, vxILight, vxIShape, vxIGeometry, vxIAppearance, vxITexture,vxIMaterial, vxIEnvironment	CSvcRqtUpdateFeature
vxIScreen, vxIWindow	CSvcRqtCreateUI
	CSvcRqtUpdateUI
vxIViewport	CSvcRqtCreateUI
	CSvcRqtUpdateUI
	CSvcRqtExportVideo
	CSvcRqtExportPicture
vxIUIControl	CSvcRqtUpdateUI

4.5.2 Example 1: Startup System

The following pseudo codes explain how to implement the function vxIRenderingSystem::StartUp:

```
SFInt vxIRenderingSystem::StartUp(const SFString& configfile);
{
   CSystemConfiguration config;
   Read the specified configuration file, and set the config object
   according to the file.
   If(file reading is failure)
       Return -1;

   m_pServerManager->Startup(config);

   CServiceRequestManager * pServiceRequestMgr;
   pServiceRequestMgr = m_pServerManager->GetServiceRequestManager();

   IServiceRequest * pRequest;
   pRequest = pServiceRequestMgr->Give(Type_CSvcRqtSystemStartup);
   Set the request according to the config object;
   pServiceRequestMgr->SendServiceRequest(pRequest);

   rtn = m_pRenderingEngine->Config(configfile);
   If(rtn){
       rtn = pServiceRequestMgr->WaitForFeedback(pRequest);
       pServiceRequestMgr->Release(pRequest);
   }
   Return rtn;
}
```

4.5.3 Example 2: Open Scene Model

The following pseudo codes show the implementation of the function vxIModel::OpenSceneModel. Here we adopt the asynchronous method to send the service request. Therefore, the return value of this function actually only means the local model is opened successfully.

```
SFInt vxIModel::OpenSceneModel(const SFString& name, SFInt mode)
{
   CSvcRqtOpenSceneModel * pRequest;
   pRequest = m_pServiceRequestMgr->Give(Type_CSvcRqtOpenSceneModel);
   pRequest->SetSceneModelName(name);
   pRequest->SetOpenSceneModelMode(mode);
   CSvcRqtCallbackOpenSceneModel pCallback = new
   CSvcRqtCallbackOpenSceneModel;
   pRequest->SetCallback(pCallback);
```

```
   m_pServerManager->SendServiceRequest(pRequest);

   Return m_pRenderingEngine->OpenSceneModel(name, mode);
}

class CSvcRqtCallbackOpenSceneModel:public ISvcRqtCallback
{
protected:
   virtual void Callback(IServiceRequest *, CServiceRequestFeedback * );
};

void CSvcRqtCallbackOpenSceneModel::Callback(IServiceRequest * pRequest,
   CServiceRequestFeedback * pFeedback)
{
   Set a flag to indicate whether the models on the render servers have
   been successfully opened or not according to pFeedback->feedback;
}
```

4.5.4 Example 3: Do Rendering and Swap Buffer

The follows pseudo codes show how to implement the functions vxIRenderingSystem::DoRendering and vxIRenderingSystem::SwapBuffer. Each screen has three rendering states, READY_FOR_RENDERING, RENDERING_IN_PROGRESS and RENDERING_IS_COMPLETED. After sending the request for doing rendering, the screen state is changed from READY_FOR_RENDERING to RENDERING_IN_PROGRESS. When the feedbacks of all of the render servers responsible to a screen are received, a callback is invoked. During the callback, the screen state is set to RENDERING_IS_COMPLETED, which means the screen is ready for buffer swapping. After buffer swapping, the screen state is back to READY_FOR_RENDERING.

```
void vxIRenderingSystem::DoRendering()
{
   For(each screen)
   {
       If(the screen is not a virtual local screen)
       {
           Let i be the screen's index;
           CScreenState& screen_state = m_pSystemStateMgr->GetScreen(i);
           If(screen_state. GetRenderingState () ==READY_FOR_RENDERING)
           {
               IServiceRequest * pRequest;
               pRequest = m_pSvcRequestMgr->Give(Type_CSvcRqtDoRendering);
               CSvcRqtCallbackDoRendering * pCallback;
```

```
                pCallback = new CSvcRqtCallbackDoRendering;
                pCallback->SetSystemStateManager(m_pSystemStateMgr);
                pRequest->SetCallback(CSvcRqtCallbackDoRendering);
                For(each viewport in screen)
                {
                    Get the camera information of the viewport;
                    Add the camera information to the request pointed
                    by pRequest;
                }
                m_pSvcRequestMgr->SendServiceRequest(screen, pRequest);
                screen_state. SetRenderingState(RENDERING_IN_PROGRESS);
            }
        }
    }
    If(there is a virtual local screen)
        m_pRenderingEngine->DoRendering();
}

class CSvcRqtCallbackDoRendering : public ISvcRqtCallback
{
protected:
    CSystemStateManager * m_pMgr;
    virtual void Callback(IServiceRequest *, CServiceRequestFeedback * );
};

void CSvcRqtCallbackDoRendering::Callback(IServiceRequest * pRequest,
    CServiceRequestFeedback * pFeedback)
{
    SFInt i = ((CSvcRqtDoRendering*)pRequest)->GetScreenIndex();
    CScreenState& screen_state = m_pMgr->GetScreen(i);
    If(pFeedback->feedback == 1)
        screen_state.SetRenderingState(RENDERING_IS_COMPLETED);
    Else
        … // handle errors
}

void vxIRenderingSystem::SwapBuffer()
{
    For(each screen)
    {
        If(the screen is not a virtual local screen)
        {
            Let i be the screen's index;
            CScreenState& screen_state = m_pSystemStateMgr->GetScreen(i);
            If(screen_state.GetRenderingState () == RENDERING_IS_COMPLETED)
            {
                IServiceRequest * pRequest;
                pRequest = m_pSvcRequestMgr->Give(Type_CSvcRqtSwapBuffer);
```

```
                m_pSvcRequestMgr->SendServiceRequest(screen, pRequest);
                screen_state.SetRenderingState(READY_FOR_RENDERING);
            }Else{
                ... // handle errors
            }
        }
    }
    If(there is a virtual local screen)
        m_pRenderingEngine->SwapBuffer();
}
```

4.6 Render Server and Server Interface

The render server works as the rendering service provider in the VCRS. Its function is to provide services to satisfy the requests sent from the render controller, and return execution results to the render controller. The render server is composed of two components:

(1) Server interface, which is in charge of the communication with the server manager on the render controller, receives service requests, calls the render engine to do the rendering and returns process results to the render controller. The rendering engine state manager describes the states of the rendering engine on the render server.

(2) For the rendering engine, please refer to the previous chapter. Fig. 4.4 shows the architecture of the render server, which can be regarded as a simplified version of the render controller.

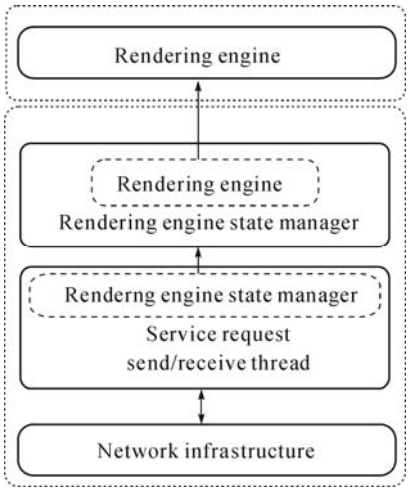

Fig. 4.4 The architecture of render server

4.7 Application: the Immersive Presentation System for Urban Planning

The Immersive Presentation System for Urban Planning (INSUP) is an application of the Visionix Cluster-based Rendering System (VCRS). The system is highly integrated and provides a toolkit for applications in urban planning, such as data acquisition, design optimization, evaluation, plan presentation and achievement, etc. The system is able to process and integrate various collected data, and manage urban-size virtual scenes. Meanwhile, the INSUP provides deep immersive integrated presentation, visualization analysis and decision assistance in the virtual environment. Figs. 4.5(a) and 4.5(b) show two snapshots of the INSUP while flying over an urban scene.

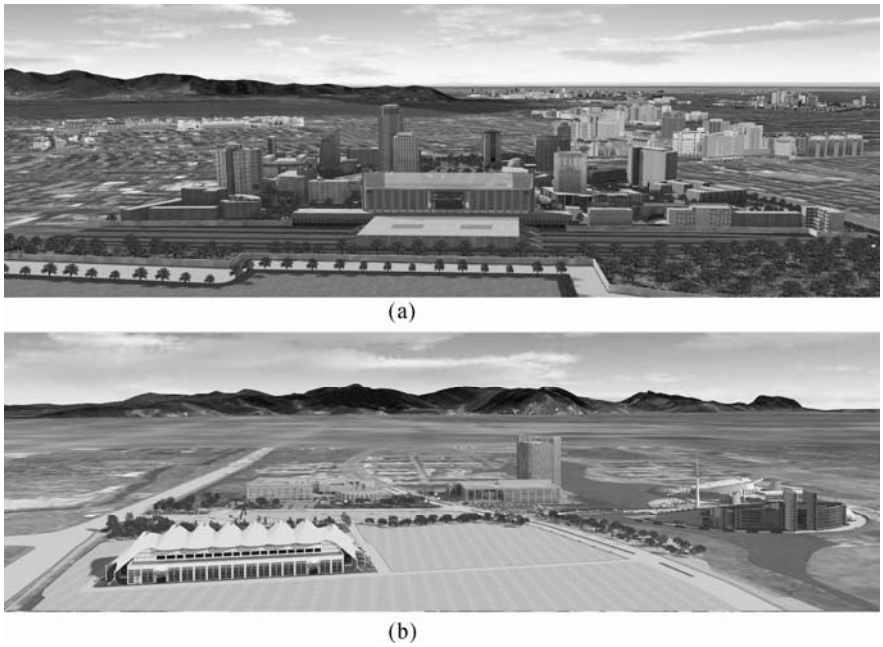

Fig. 4.5 Snapshots of the INSUP while flying over an urban scene. (a) A bird's eyes view of a railway station; (b) A bird's eyes view of a university campus

The INSUP is based on the VCRS. It implements the computation flexibility architecture and is capable of scheduling TB-size scene data. Thus, the INSUP can render a complicated scene which contains tens of millions of triangles in real time. The INSUP successfully simulates reflection, refraction, transmission and shadow, optical phenomena, etc. The system is able to present the natural geographical environment including buildings, terrain, plants, weather, water, etc. Figs. 4.6(a) – (f) show some visual effects provided by the INSUP.

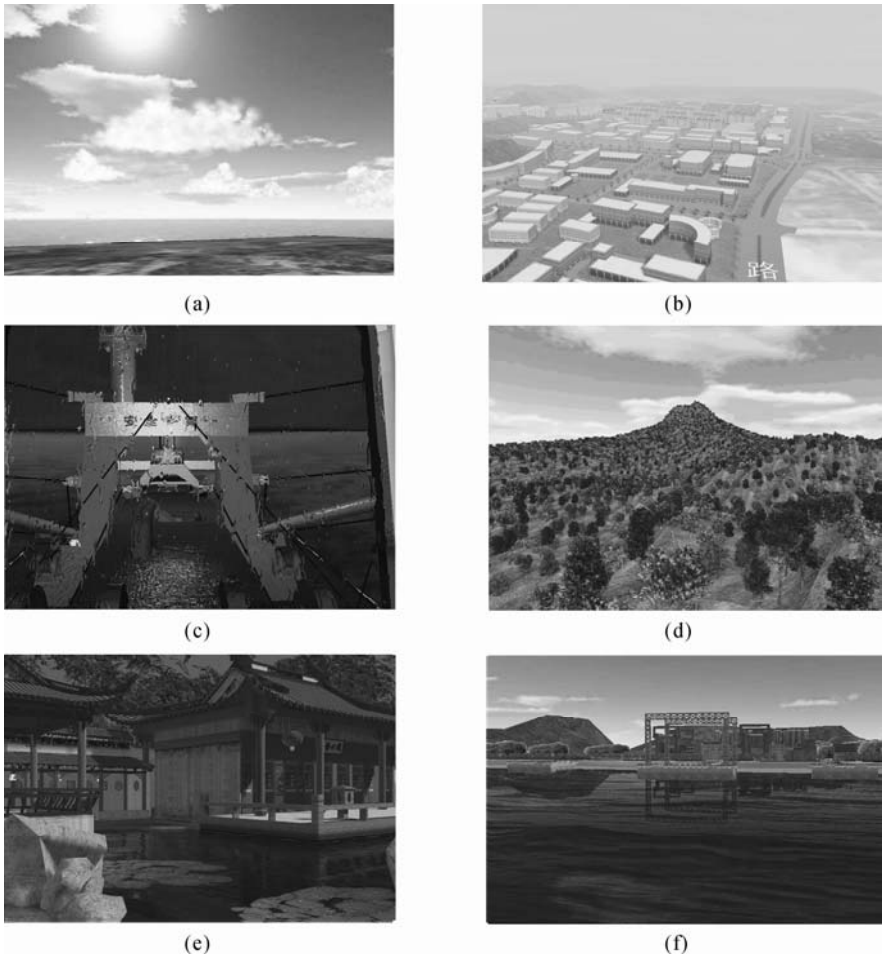

Fig. 4.6 Some visual effects provided by the INSUP. (a) Sun, clouds and sky; (b) Fog; (c) Rain; (d) Terrain and vegetation; (e) Building and pond; (f) Harbor and ocean

The INSUP supports the synchronized rendering of up to dozens of display channels, and thus implements ultra resolution display. For the projection type, the INSUP supports different topologically and geometrically arranged display screens, and displays in many ways regular/irregular projected screens. For the integrated area of multi projection, the INSUP implements the seamless integration of the pixel position, chroma and illuminance. With the 3D tracker, data glove, touch screen, microphone, human-computer interaction devices, etc., the INSUP is able to present a 360° panorama 3D stereo interactive display. Fig. 4.7 shows the display screen of the 360° multichannel stereo display environment.

Fig. 4.7 The display screen of the 360° multichannel stereo display environment

4.7.1 System Deployment

- *Hardware Platform*

The self-designed graphics processing cluster works as the computing hardware. The multi-projector display system works as the main output display. The 3D tracker, data glove, touch screen, microphone, etc. work as the user input devices.

At the beginning of system development, we designed a 6 channel stereo display environment to be the display platform. The system contains a 6 m×1.8 m orthographic cylinder screen, and employs 6 projectors to build the passive stereo projection system in a polarized manner. The 6 projectors are divided into two groups, and each group of 3 projectors forms a completed single eye (left/right) arch window. The viewing angle at the center of the arch is 90°. The computation platform was arranged into a 10-node cluster. Fig. 4.8 shows a harbor scene displayed by the 6 channel stereo display system.

Fig. 4.8 A harbor scene displayed by the 6 channel stereo display system

In 2008, the 6-channel system migrated into our self-designed 360° multichannel stereo display platform. The system contains a 6m-diameter cylinder screen and employs 14 projectors to display the seamlessly integrated 360° scene, shown in Fig. 4.7.

- *Software Platform*

The main program of the INSUP runs on the render controller node. The controller node contains a virtual local screen that provides the local user interface. Fig. 4.9 shows the user interface of the INSUP on the render controller.

Fig. 4.9 The user interface of the INSUP on the render controller

There are 14 render servers that provide render service for the 360° cylinder screen. In order to implement the multi-projection seamless integration, each render server deploys a multi-display image corrector. To provide various ways of human-computer interaction, we developed voice/gesture recognization, touch screen, etc. interaction models.

4.7.2 Functionality

The INSUP provides a deep immersive platform for evaluation and presentation. The main functionalities include:

(1) Providing a 360° panorama real-time display and urban space travel.

(2) Human-computer interaction via gesture, voice and touch, etc. Figs. 4.10(a), (b) and (c) show the three types of human-computer interaction in INSUP respectively.

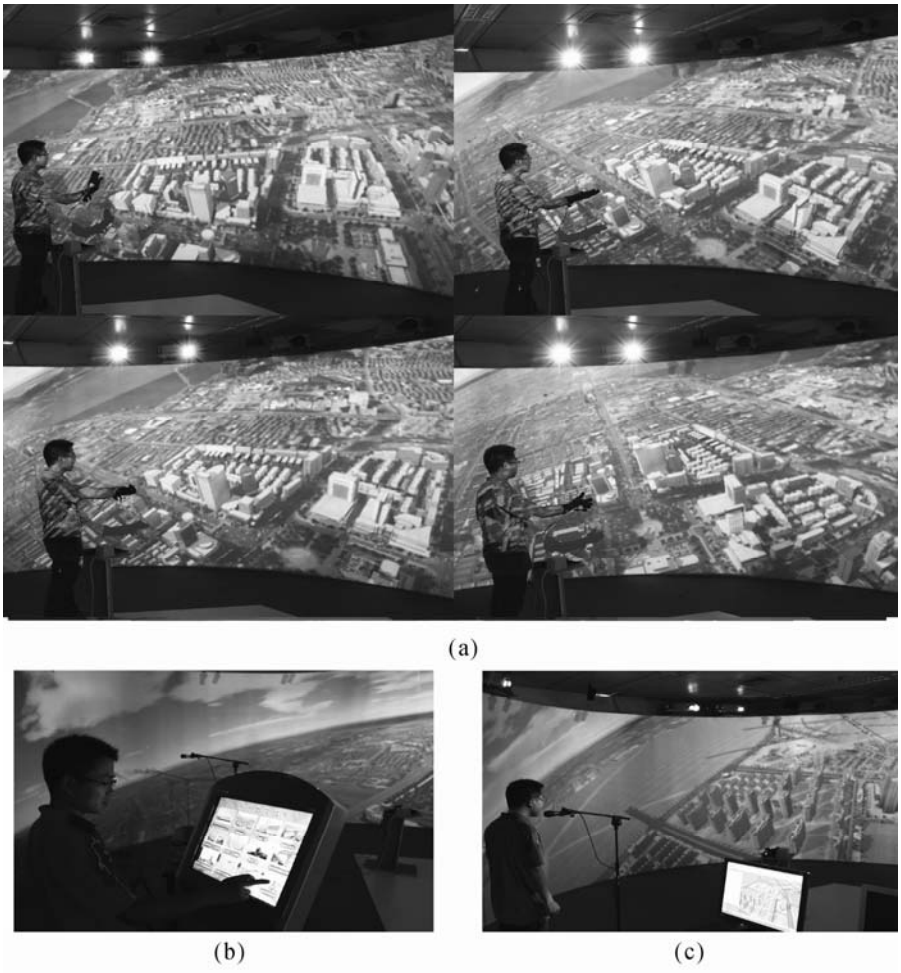

Fig. 4.10 The multimodality human-computer interaction in INSUP. (a) Interaction through gesture; (b) Ineraction through touch; (c) Interaction through voice

(3) Building block design during planning.

(4) Providing multi-plan comparison and decision making support via multi-display windows in the 360° cylindrical display environment. Fig. 4.11 shows a comparison between two urban plans. Each urban plan is rendered under either day or night lighting conditions.

Fig. 4.11 The comparison between two urban plans, each of which is rendered under either day or night lighting conditions

(5) Providing visual operation tools, such as spatial measurement, view line analysis, etc. The snapshots for the two visual tools are shown in Figs. 4.12(a) and 4.12(b).

Fig. 4.12 Two visual tools. (a) Pipeline measurement; (b) View line analysis

References

Molnar S, Cox M, Ellsworth D, Fuchs H (1994) A Sorting Classification of Parallel Rendering. IEEE Computer Graphics and Applications, 23-32

Montrym JS, Baum DR, Dignam DL, Migdal CJ (1997) InfiniteReality: a real-time graphics system. Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques

Staadt OG, Walker J, Nuber C, Hamann B (2003) A Survey and Performance Analysis of Software Platforms for Interactive Cluster-Based Multi-Screen Rendering. Eurographics Workshop on Virtual Environments

5

Optimal Representation and Rendering for Large-Scale Terrain

With fast developments in remote sensing technologies, 3D scanning and image-based modeling methods, the available datasets for large-scale scenes, such as terrains and vegetation, have exploded in recent years. How to represent and render these large-scale data has always been an important problem for applications in virtual reality, 3D interactive design and real-time simulation. The size of the large-scale data is so great that it far exceeds the capacity of standard graphics hardware. It is impossible to interactively visualize them without the support of optimal representation and rendering algorithms. Therefore, the challenge for researchers is to design well-suited data structures and efficient rendering algorithms, to make the users interactively view and edit the large scale datasets.

In this chapter, the representation and rendering for large-scale terrains are introduced, which is one of the main components in the rendering of virtual natural scenes. The geometry and texture of natural scenes are so diverse and irregular that it is difficult to effectively describe them. Furthermore, a huge amount of data is needed to represent natural scenes with high resolution. So it is very difficult to render high quality natural scenes in real time. Fortunately, according to the vision research results, the discriminability of human eyesight is limited, which makes the user unable to attend to all parts of the scene. The method introduced in this chapter makes the following observations: Firstly, the natural scenes possess a very wide field of view. Instead of paying attention to some details, the user usually attends to the whole rendering effect. Secondly, the positional relationships of geometrical faces in natural scenes are so complex that they are often interleaved or inter-covered (such as forests or clouds). This phenomenon allows some errors in the faces' location, since human eyes with limited discriminability find it difficult to notice such errors. The key problem in our methods is to find a reasonable balance and transition between whole scenes and local details. We propose a scheme for sampling the natural scenes to build an approximation model of the original data. Rendering from this model, we can get an approximation of the original natural scene. By adjusting the parameters to

change the approximating precision, we get a desirable rendering result with a balance between the rendering quality and speed.

Real-time rendering of a large-scale terrain plays a key role in many applications, such as 3D games and simulators. In this section, we first give an overview of recent terrain rendering methods. Then we describe our procedural terrain rendering algorithm.

5.1 Overview

In recent years, many researchers have done substantial research on real-time large-scale terrain rendering. They usually adopt two kinds of approaches to solve this problem. One is to build the level of detail (LOD) model of the terrain and generate a view-dependent mesh from the LOD model during rendering. The other is to perform multilevel paging for the massive data of the LOD terrain model, dynamically loading the requested data from external storage to the memory.

5.1.1 LOD Model of Terrain

In this kind of method, the procedure is divided into preprocessing and rendering stages. In the preprocessing stage, the continuous level of detail model is built for the whole terrain, from the coarsest level to the finest level. In the rendering stage, the appropriate level of detail is selected according to the viewpoint and a view-dependent adaptive grid is built. Such a grid is finer in places near the viewpoint, or with high undulation, and coarser in places far from the viewpoint or when flat.

There are two kinds of grids used in the LOD terrain model. One kind consists of triangulated irregular networks (TIN) and the other is a regular square grid (RSG). The TIN produces fewer triangles, but with more complexity, and easily produces slim triangles. The RSG has simple structures and is favorable for error computation and fast rendering.

5.1.1.1 Triangulated Irregular Networks (TIN)

Hoppe H (Hoppe, 1997) presented a progressive mesh, which was later improved and applied in terrain rendering to build TIN (Hoppe, 1998). In this approach, the terrain is divided into equal-sized blocks. For each block a progressive mesh is built. This method uses fewer triangles to build terrain grids. But the procedure to build the grid is more complex and would cost more time.

Toledo R *et al*. (Toledo *et al*., 2001) extended Hoppe's work by combining the regular grid and irregular progressive grid, which is called the QLOD model. The QLOD model is composed of a quadtree, where each node corresponds to a terrain block with a progressive mesh. We can easily perform fast view culling and build the grid within the error threshold for this structure.

In the TIN models, the selection of triangle vertices is more flexible, which gives us a more optimized grid with fewer triangles than those obtained with the RSG model under the same error control. However, the TIN grid construction requires more memory, more computation time and a more complex data structure. It is also very difficult to page in terrain data from external storage to the memory with the TIN grid. So the RSG model is more popular in large-scale terrain visualization systems.

5.1.1.2 Regular Square Grid (RSG)

For the methods using the RSG model, the original terrain data is treated as a uniformly sampled height field, for which a hierarchical tree of regular triangles, usually a quadtree or a binary tree, is built. The tree is traversed and the triangle strip is built during rendering. Although the triangles produced here are more than those produced by the TIN model, this kind of method can perform faster and easier view culling and bottom-to-top simplification. Besides, the connectivity of triangle vertices is implied in the regular grid, which makes the data more compact and easier to be paged in from the external storage.

- *Quadtree-based Adaptive Triangle Mesh*

To build a quadtree, the middle points of four sides of the height field are connected to get four sub-regions. By recursively dividing these four sub-regions, the quadtree is built where each node corresponds to a terrain block. The key problem in this kind of approach is how to stitch the adjacent terrain blocks seamlessly when they are of different resolutions.

Herzen BV and Barr AH (Herzen and Barr, 1987) first proposed a method for building quadtree-based adaptive triangle meshes and applied it to parametric surfaces. In order to avoid the seams between triangles, they proposed a restricted quadtree, which requests the adjacent nodes to only differ by one level of detail. Although this method solves the seam artifacts, the structure of this model is not compact and this model contains many unnecessary triangles.

Lindstrom P *et al*. (Lindstrom *et al*., 1996) designed a real-time terrain rendering algorithm-based on quadtree representation. By allowing triangles to be recursively combined from bottom to top when the projected image error is less than the screen-space threshold, a mesh with fewer triangles can be used to represent the height field. To guarantee that there is no seam between adjacent triangles, the inter-vertices dependencies need to be recorded. The advantage of

this algorithm is that by computing the view-dependent screen error to choose the appropriate triangles to build a triangle mesh, the rendering error is controllable and the rendering achieves a good balance in rendering quality and speed. The disadvantages are that extra data are needed to record the vertices dependency to avoid the seams and larger memory is requested to store the finest grid to support the bottom-up simplification.

Restricted Quadtree Triangulation (RQT) (Pajarola, 1998) was presented to render a large-scale terrain. This method effectively reduced the data redundancy in the method of Lindstrom P *et al*. (Lindstrom *et al*., 1996) and cost less memory. A saturated error metric is introduced to remove the dependencies in the triangle hierarchy. Compared with Lindstrom P *et al*. (Lindstrom *et al*., 1996), this method did not need to explicitly express the quadtree. Only the digital heights and errors in the terrain sampling points need to be recorded, which makes the data size smaller.

The above methods build dynamic adaptive meshes during rendering. More time is needed to render finer meshes when higher rendering quality is demanded. Ulrich T (Ulrich, 2002) proposed a structure called Chunked LOD. This method consists of first building the static LOD model of the terrain and then building the quadtree structure, where each node corresponds to a block of terrain mesh. The node is directly used as a rendering unit to simplify the mesh building procedure. Since the node terrain mesh is pre-built, the connectivity between nodes can also be solved in the preprocess. This avoids the seam problem. The problem with this method is that the pre-built static mesh is not necessarily the optimal mesh.

- *Binary-Tree-Based Adaptive Triangle Mesh*

Duchaineau MA *et al*. (Duchaineau *et al*., 1997) did pioneering work on a bintree-based terrain grid rendering by proposing the ROAM (Real-time Optimally Adapting Meshes) algorithm. This method effectively avoided the T-connection by forcing the triangles to split. The ROAM algorithm selects the appropriate triangle nodes from the bintrees by view-dependent screen-space bounds. In order to utilize the frame-to-frame coherence, they used two priority queues to drive split and merge operations. This allows the algorithm to adjust the triangles in the two queues from the previous frame. Since the number of triangles that are changed per frame is not so large, the update is very efficient. For each frame, the priorities of all triangles need to be computed. When there are many triangles in the queues for higher-resolution terrains, the computation of priorities will cost a lot of time and severely influence the rendering speed.

In order to further improve the efficiency of the ROAM algorithm, researchers propose many improvements. When the terrain mesh to be built is of a high resolution, the mesh-building procedure costs a lot of CPU computation time. Pomeranz A (Pomeranz, 2000) optimized this procedure by using triangle clusters as the building unit. The number of triangles produced by this method is more than the ROAM's. However, the mesh is finer than ROAM's and it is easy to construct a triangle strip for such meshes. Levenberg J (Levenberg, 2002) proposed CABTT

(Cached Aggregated Binary Triangle Trees) to save computation time. Different from Pomeranz A (Pomeranz, 2000), CABTT stored the triangle clusters used by continuous frames in the host memory and made the rendering faster. However, this algorithm had a large dependency on the viewpoint's coherency. If the position of the viewpoint changes too much, the triangle clusters stored in the host memory will be useless and the rendering speed will become slower.

In ROAM, the positions of the sub-triangles and adjacent triangles need to be recorded in order to access them, which will consume some memory space. Evans F *et al*. (Evans *et al*., 1996) and Evans W *et al*. (Evans *et al*., 1997) proposed Right Triangulated Irregular Networks (RTIN), in which the triangle bintree is efficiently constructed to save data redundancy. This algorithm marked the left and right sub-triangles as 0 and 1. In this way, each triangle of the bintree is represented by a binary list, which is called vertex indices. By this encoding, only the height value and vertex index need to be recorded for each triangle. The sub-triangles and adjacent triangles can be accessed by the vertex indices, which effectively saved the data space.

- *Hierarchical Patch-Based Terrain Mesh*

In this kind of method, the terrain is divided into multi-resolution rectangle patches, which is called the hierarchical terrain patch. During rendering, multiple terrain patches are selected. One terrain patch has the same resolution, while different terrains may have different resolutions. This kind of method does not pay attention to accurate mesh building, or view-frustum culling. Bishop L *et al*. (Bishop *et al*., 1998), Rottger S and Ertl T (Rottger and Ertl, 2001), Platings M and Day AM (Platings and Day, 2004) and Pi X *et al*. (Pi *et al*., 2005) utilize the parallel computing ability of modern graphics hardware and achieve high rendering efficiency. How to remove the cracks between the terrain patches of different resolutions becomes the main problem for this kind of method.

Boer WH (Boer, 2000) presented the GeoMipMap (Geometrical MipMap) based on the quadtree terrain structure. The node of the quadtree corresponds to one terrain patch. The terrain patches on the same level of the quadtree have the same resolution. During rendering, the GeoMipMap is traversed according to the viewpoint and multiple appropriate terrain patches are selected to construct the whole terrain mesh. However, cracks will appear when the adjacent terrain patches have different resolutions. To solve this problem, the author modified the adjacent terrain patches with higher resolution by removing some triangles on the terrain border. This algorithm is simple and easy to implement. The triangles included in the terrain patches are regular and easy to strip, which makes it convenient for the graphic processor unit to perform parallel processing. However, the triangles in the higher resolution terrain patch are important for rendering the finer terrain mesh. Reducing these triangles would impair the rendering quality.

By adding triangles in the lower resolution terrain mesh, Wagner D (Wagner, 2004) removed the cracks between patches and kept the high resolution terrain untouched. Whether adding more triangles in the low resolution terrain mesh or

reducing triangles in the high resolution terrain mesh, such computation would consume CPU resources. To effectively reduce this computation time, Pouderoux J and Marvie JE (Pouderoux and Marvie, 2005) proposed a terrain strip mask. He divided the terrain into equal-sized patches, which are called regular tiles. To reduce the cracks between terrain patches with different resolutions, a set of strip masks are constructed to record the vertices connections. During rendering, the proper strip mask is selected for each regular tile to build the terrain mesh.

Most terrain rendering algorithms need to pre-build the LOD model of the terrain. These LOD models usually contain extra information for rendering and will occupy some memory space. Losasso F and Hoppe H (Losasso and Hoppe, 2004) used the terrain geometry as texture and imported a clipmap structure in this algorithm to propose the Geometry Clipmap. Different from previous methods, this algorithm uses a hollow rectangle as terrain patch, which is divided into four triangle strips. To remove the cracks between rectangles of different resolutions, the authors adopt a perpendicular triangle to amend cracks between neighborhoods. Using this method, we can adopt texture compression to compress terrain geometry and decompress the texture during rendering.

5.1.2 Out-of-Core Techniques

A large-scale terrain contains massive geometric data, which is too huge to fit into the main memory. We need to page in the data between the external storage and main memory according to the rendering requests, which we call out-of-core rendering techniques, an effective way to realize large-scale data real-time rendering. Since the data being transferred between external storage and memory is much slower than the data being transferred between the memory and the GPU, this is the bottleneck of the out-of-core technique. Such algorithms must be optimized to efficiently fetch and access data stored in hard drives.

To enhance the rendering efficiency, the terrain data need to be reorganized to ensure continuity and locality (Gotsman and Rabinovitch, 1997; Bajaj *et al*., 1999; Pascucci, 2000; Vitter, 2001). These algorithms did not take into account data compression, so the data to be transferred are large in size. Gerstner T (Gerstner, 1999) reduced the paged data size by encoding the terrain data. He organized the terrain grid as a bintree structure and performed index encoding for triangle vertices. The vertices were stored in a one-dimensional array and were easy to page in from external storage. However, this algorithm restricted the order to just paging in the data, which made it impossible to access the data in an arbitrary order. In order to optimize the paging order, Lindstrom P *et al*. (Lindstrom *et al*., 1997), Lindstrom P and Pascucci V (Lindstrom and Pascucci, 2001; Lindstrom and Pascucci, 2002) studied how to improve the consistency between the data in the memory and the data in the external storage, so as to reduce the number of paging events from external storage to main memory. He proposed a method to

produce the layout of the triangle vertices, restoring the data according to the order of filling curves in ∏ space.

Although, in the algorithm of Lindstrom, multiple triangle vertices were read in at one time, a single triangle was still used as the unit for building the terrain mesh. Cignoni P *et al*. (Cignoni *et al*., 2003a) combined a batch of triangle-reading operations with the mesh building, to propose BDMA. They assembled adjacent triangles to make triangle clusters and adopted the TIN structure. During rendering, they used the triangle clusters as the unit to page in data and build a mesh, which further improved the efficiency of the out-of-core technique.

In summary, out-of-core achieves real-time large-scale terrain rendering by multilevel paging. On a computer with a single CPU, when the viewpoint changes a lot, some time is needed to wait for the data to be paged in from the external storage, which will lead to the frame stagnancy artifact. Therefore, many researchers designed out-of-core algorithms in multi-thread format to run on computers with multiple CPUs, where the out-of-core technique provides more advantages.

5.2 Procedural Terrain Rendering

As we have introduced in the previous sections, the out-of-core technique is proposed for transfering large-scale terrain data between host memory and external storage (Lindstrom and Pascucci, 2002; Cignoni *et al*., 2003b). However, this technique only addresses the problem of how to render the large-scale terrain in a single computer, without any consideration of visualizing the remote terrain models through the network. In addition, the LOD models of terrains described in previous sections are only designed for direct rendering. When the digital elevation models (DEMs) of terrains are modified, the LOD models need to be rebuilt, which is very time-consuming and inconvenient.

In many applications, such as virtual emulators and video games, the accuracy of the height values of terrain data is not so important. We can compress the terrain data here with a high compression ratio to greatly reduce the terrain data size. As was shown by Ebert DS *et al*. (Ebert *et al*., 1998), the fluctuation of the terrain in the real world has self-similar fractal characteristics. So the fractal compression algorithm can be adopted to compress the terrain to a small and compact data set. The compressed terrain data will be small enough to be transferred through the network with little latency.

How to simulate the natural terrain by fractal parameters has been studied in-depth by many researchers. Mandelbrot BB (Mandelbrot, 1975) introduced fractional Brownian motion (fBm) and described the random fractal characteristics in a natural scene. Fournier A *et al*. (Fournier *et al*., 1982) proposed a recursive subdivision algorithm to estimate fBm, and obtained the rough terrain geometry.

Lewis JP (Lewis, 1987) presented a spectral modeling approach based on fBm to achieve more realistic terrain geometry. Kaplan LM and Kuo CJ (Kaplan and Kuo, 1996) applied Fourier transformation to fBm calculation to obtain more accurate self-similarity image results. Szeliski R and Terzopoulos D (Szeliski and Terzopoulos, 1989) developed a constrained fractal approach to represent terrain, which is a hybrid of splines and fractals, but the iterative procedure cost much computation time. Musgrave FK *et al*. (Musgrave *et al*, 1989) and Nagashima K (Nagoshima, 1997) introduced hydraulic erosion and thermal weathering into the fractal terrain to improve the reality of a computer- generated terrain. All of the above approaches generate the terrain geometry offline, and take no account of terrain rendering.

Procedural terrain rendering is a new idea which integrates terrain generation and rendering. Losasso F and Hoppe H (Losasso and Hoppe, 2004) dealt with the terrain geometry as the texture and introduced the geometry clipmap to manage the terrain in a set of nested regular grids. They also synthesized the terrain by fractal noise. However, the whole terrain is determined by a single fractal parameter, which makes the rendering result not so pleasing. Dachsbacher C and Stamminger M (Dachsbacher and Stamminger, 2004) described the approach of adding procedural geometry to the digital elevation map on the fly. The sketch terrain heights are stored in the geometric image by Gu X *et al*. (Gu *et al*., 2002), to which detailed fractal noises are added during the rendering stage.

Different from the traditional terrain rendering approaches, we propose a procedural terrain rendering technique by fractal approximation. Our approach adopts a fractal parametric model to approximate the original terrain and, only at the rendering stage, the appropriate details of the height field are generated from the fractal parametric model by GPU, and further formed into the view-dependent mesh. The difference between the original terrain meshes and the generated one is controlled by a user-specified threshold. Since the size of the fractal parametric representation is much smaller than that of the original height field, compared with those out-of-core approaches, the fractal parametric model with the same data size covers a much wider terrain region, which effectively reduces the transmission amounts between external storage and host memory. Therefore, our method achieves steady high performance on the computer with a single CPU. Furthermore, after the fractal model is edited by the user, such as synthesizing a new terrain, it can be reused for rendering directly, without the complex construction procedure requested by the LOD model. In addition, our approach can add procedural details to the original terrain to improve the rendering quality, which is different from the previous compression approaches. Our main contributions include:

(1) A parametric fractal model to approximate the original terrain and to reduce the storage greatly.

(2) A view-dependent error estimation algorithm, based on probability, to choose the proper detail levels for terrain patch generation.

(3) A generation approach to the terrain model by GPU to decrease the transferred data between host and video memory.

5.2.1 An Overview of Asymptotic Fractional Brownian Motion Tree

In this section, we will first introduce our idea and then the fBm-tree structure.

5.2.1.1 Our Idea

Fractional Brownian motion (fBm), a kind of fractal noise, has been applied to terrain generation (Fournier *et al*., 1982; Ebert *et al*., 1998). It is difficult to approximate the complex terrain with the fractal model by a single fBm parameter. For example, the original terrain is illustrated in Fig. 5.1(a), and the terrain generated by only one fBm parameter is shown in Fig. 5.1(b), which is very different from Fig. 5.1(a). Therefore, we must introduce more information to control the shape of the terrain generated by the fractal model.

In order to control the difference between the generated terrain and the original one, we build the base mesh with important height values by sampling the original terrain. The approximate terrain is produced by adding the fractal noise into the base mesh (Fig. 5.1(c)). To generate the view-dependent terrain mesh, we construct the quad-tree structure at the preprocess stage and incorporate the fractal information with base mesh in the rendering stage. The appropriate fractal parameters and height values are selected by traversing the quad-tree according to the given threshold. In the next paragraphs, we introduce the fractal model based on fBm.

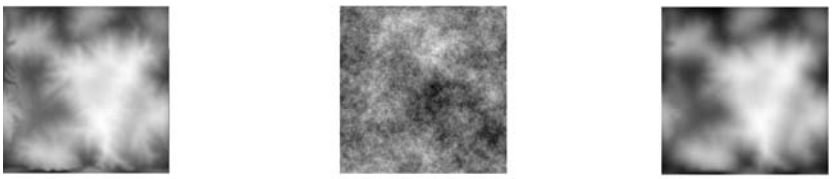

Fig. 5.1 Comparison between original terrain and generating one. (a) The gray image of the original terrain; (b) The generating result where only one fractal parameter is different from the previous one; (c) The terrain using our afBm-tree is close to the original one

The roughness parameter of fBm is the key factor in controlling the fluctuation of the terrain. Since the roughness factor is fixed in fBm, the fluctuation of the generated terrain remains invariant under any resolutions. However, even though the fluctuation of the real terrain seems consistent in a wider range, it is different for a shorter range. A general fractal model, called asymptotic fBm (afBm) (Kaplan and Kuo, 1996), is proposed to simulate the natural terrain more accurately. The roughness parameter in afBm is influenced by the sampling interval.

The fBm denotes a family of Gaussian stochastic processes in a self-similarity condition for many natural time series (Fournier *et al*., 1982). The fBm in 2D is

shown in Eq.(5.1):

$$Var\left[F(x+d_x, y+d_y)-F(x,y)\right]=\sigma^2\left|d_t\right|^{2H} \tag{5.1}$$

where $d_t=\sqrt{d_x^2+d_y^2}$ and H is known as the Hurst parameter ($0<H\leq 1$). The less H is, the more rugged the generation terrain becomes. The function $Var(x)$ is the statistical variance of a random variable x, and σ^2 is a constant which is equal to the variance when d_t is 1. $F(x, y)$ can be regarded as the height function at the position (x, y) in the height field. Kaplan LM and Kuo CJ (Kaplan and Kuo, 1996) generalized the fBm model to represent objects whose roughness is scale-dependent. It can be shown in 2D form as Eq.(5.2) :

$$Var\left[F(x+d_x, y+d_y)-F(x,y)\right]=\sigma^2 f(d_t) \tag{5.2}$$

where the function $f(t)$ is called the structure function. The structure function $f(t)=|d_t|^{2H}$ defines the process known as the fBm. A more complex structure function, called asymptotic fBm (afBm), is proposed in this paper. Eq.(5.3) shows the structure function in afBm:

$$f(t)=(1-A)\frac{\rho^{|t|}-1}{\rho-1}+A\,|\,t\,|^{2H} \tag{5.3}$$

where $0\leq\rho<1$ and the constant A is the smoothness parameter. We use the afBm function to represent the roughness of the terrain height for more realistic simulation.

5.2.1.2 Our Representation and Overview

In this subsection, we use afBm to construct the fractal parametric model, which is called an afBm-tree. Our approach extracts an afBm-tree from the original terrain in the pre-processing stage, and generates the view-dependent terrain meshes during rendering. The structure of the base mesh is regular, to simplify the representation. Each afBm-tree node corresponds to a patch of the terrain region, and records the afBm parameters and the base height. The afBm parameters represent the statistical roughness of the corresponding terrain region and the base height records the height values of four corners of the terrain region. Each node has a level index, denoted as l, and the level index of the root node is 0.

The afBm-tree approximates the original terrain with fractal information and the approximating accuracy is controlled by the user-defined threshold. During the rendering stage, the height field patch generated from an afBm node is called a

fractal patch, as is illustrated in Fig. 5.2. Each afBm node can generate fractal patches of multi-levels. For the k-th detail level, the resolution of the fractal patch is $(2^k+1)\times(2^k+1)$ $(0\leq k\leq k_{max})$, where k_{max} is the maximal level specified by the user. The fractal patch can be stored in the video memory as textures out of its regularity.

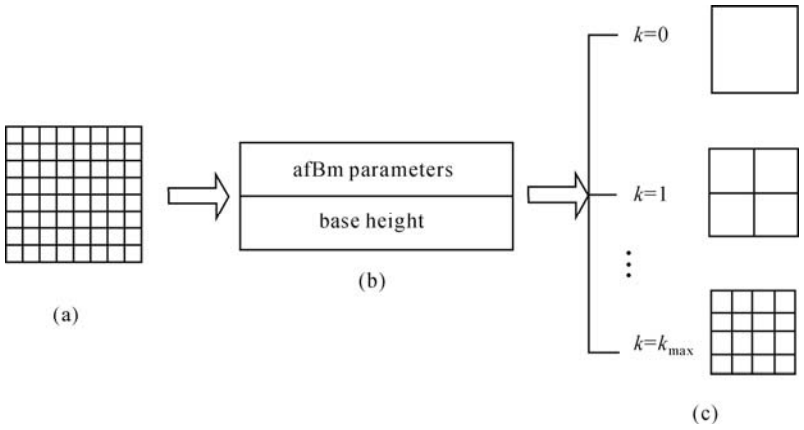

Fig. 5.2 Fractal patch. (a) Original terrain region; (b) afBm node; (c) Fractal patch

Since triangle strips are more efficient to render than indexed triangles, we use preprocessed mesh, called connecting template, to render the terrain. It contains a set of 2D triangle strips which form a square, as shown in Fig. 5.3. They only specify the connection of the vertices and the height values are retrieved from the fractal patches by GPU.

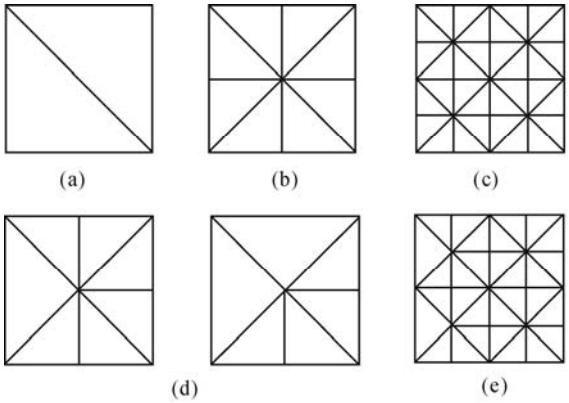

Fig. 5.3 Connecting template. (a) $level_0$; (b) $leve_l$; (c); $level_2$; (d) Two incomplete connecting templates of $level_1$; (e) An incomplete connecting template of $level_2$

The connecting templates of different levels are illustrated in Figs. 5.3(a), 5.3(b) and 5.3(c). They are complete because they use all vertices of the fractal patches. However, if we only use these complete connecting templates to render

terrains, the crack artifacts would occur when the connecting templates of different levels are connected. To avoid this problem, proper incomplete connecting templates should be designed. Fig. 5.3(d) illustrates two incomplete connecting templates of $Level_1$, which make transitions between connecting templates of $Level_0$ in Fig. 5.3(a) and those of $Level_1$ in Fig. 5.3(b). Another incomplete connecting template of $Level_2$ is illustrated in Fig. 5.3(e). All these complete and incomplete connecting templates are stored in the video memory in the format of display lists.

Our approach includes two stages: afBm-tree construction and procedural terrain rendering (Fig. 5.4). In the pre-processing stage, the afBm-tree is constructed from the original height field recursively. The subtree of the afBm-tree satisfying the defined precision is selected at the initialization of the rendering. During rendering, the afBm subtree is traversed and the correct afBm nodes are selected. The fractal patches satisfying the selection criterion are generated according to the current viewpoint. The corresponding connecting templates and the fractal patches are sent into the GPU to render the final terrain mesh.

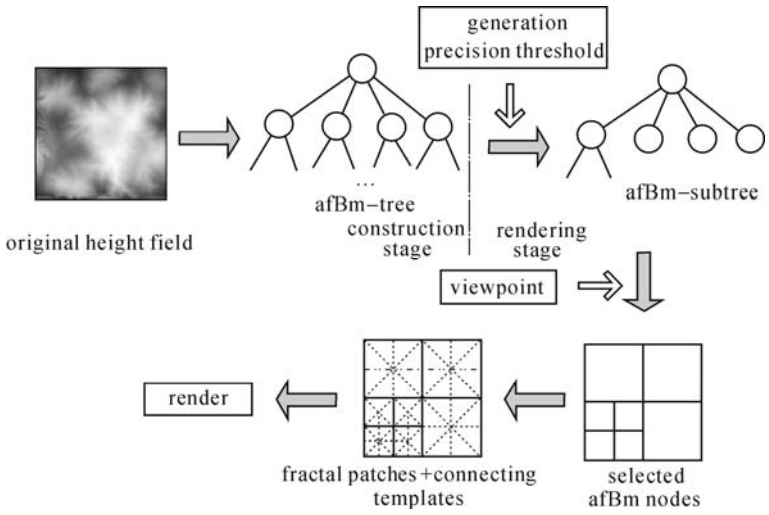

Fig. 5.4 The rendering pipeline

5.2.2 afBm-Tree Construction

We build the afBm-tree for the whole terrain, of which each node records the fractal information of a patch of the terrain region.

Assume the size of the original height field to be $(2^M+1)\times(2^M+1)$. The quad-tree is built by subdividing the terrain recursively. The root node of level 0 corresponds to the whole height field, and the node of level i corresponds to the region with the size of $(2^{M-i}+1)\times(2^{M-i}+1)$. The minimal size of a terrain region

relevant to a node is $(2^m+1)\times(2^m+1)$, $1\leqslant m<M$. We compute the afBm parameters for the quad-tree nodes of level i, where $i = 0, 1,\cdots, M-m$.

The base height is used to control the shape of the generated fractal patch from the afBm-tree node, and consists of four height values of the corners of the terrain patch, shown by the blue points in Fig. 5.5(a). There are four afBm parameters to be calculated, H, σ^2, ρ and A. Firstly we estimate H and σ^2. From Eq. (5.1) and the statistical property of the fBm, we deduce:

$$\ln E([F(x+d_x, y+d_y)-F(x,y)]^2) = \ln\sigma^2 + (2\ln|d_t|)H \tag{5.4}$$

where $E(t)$ is the expectation function. Thus, we collect the data pairs of $\{\ln|d_t|, \ln E([F(x+d_x, y+d_y)-F(x,y)]^2)\}$, as shown by the red points in Fig. 5.5 (b). H and σ^2 can be calculated by a linear regression algorithm and the corresponding line is shown in Fig. 5.5 (b). When H is estimated, we could calculate ρ and A by the method of Kaplan LM and Kuo CJ (Kaplan and Kuo, 1996).

After all terrain regions are processed in top-down order, the afBm-tree is constructed (Fig. 5.5(c)).

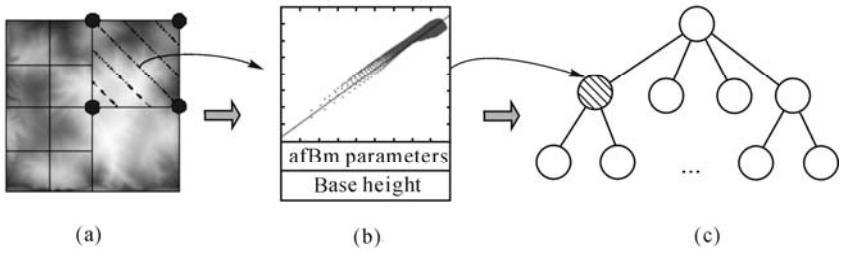

Fig. 5.5 The procedure of afBm-tree construction. (a) Terrain region; (b) afBm calculation; (c) afBm-tree

The value of σ^2 specifies the difference between the original terrain and the generated one from the statistical point of view. The greater the value of σ^2 is, the more different the generated terrain is from the original one. It is used to control how the generated terrain approximates to the original one. To select afBm nodes correctly, the value of σ^2 is further adjusted in a saturated form:

$$\sigma_{T_i}^2 = \begin{cases} \sigma_{T_i}^2, & i = M-m, \\ \max(\sigma_{T_i}^2, \sigma_{T_{i+1}}^2), T_{i+1}\subset T_i, & 0\leq i < M-m \end{cases} \tag{5.5}$$

where $T_{i+1}\subset T$ denotes that T_{i+1} is the child node of T_i.

5.2.3 Procedural Terrain Rendering

5.2.3.1 afBm Subtree Extraction

Before rendering, we need to extract the afBm subtree from the pre-computed afBm-tree according to the user-specified threshold Ω, which specifies the allowed difference between the original terrain and the generated one.

The afBm tree is traversed in top-down order recursively. For the node T, if the generation error $\sigma_T^2 \le \Omega$, the node T satisfies the generation precision requirement. Otherwise, its child nodes are visited recursively. After the traversal is finished, the afBm subtree is constructed. The afBm subtree is utilized to create the view-dependent terrain meshes during rendering.

5.2.3.2 View-Dependent Error Estimation

The rendering algorithm needs to estimate whether the node T of the afBm-tree can generate the terrain mesh that satisfies the rendering requirements. However, the real height values are unknown before terrain generation, so the accurate error values are unavailable for estimation. Therefore, we adopt a maximum likelihood method to estimate the error according to the fractal parameters.

To control the height range, we define the fractal-box for node T, just similar to the conditional bounding box. The 2D projection of the fractal-box is equal to the terrain region of the afBm node T, as shown in Fig. 5.6(a). Assume the base height of T is $F(x_i, y_j)$ $(i, j = 0, 1)$. Given the position (x_p, y_q) in the region of T, where $0 \le p,q \le 1$, according to the theory of afBm, the height at this position is:

$$F(x_p, y_q) = \bar{F}(x_p, y_q) + \sum_{k=0}^{k_{\max}} \Delta_k \cdot N_{\text{gauss}} \tag{5.6}$$

where $\bar{F}(x_p, y_q)$ is the bilinear interpolation value of the base height. The value of Δ_k is decided by the fBm parameters deduced from Eq. (5.1) as follows:

$$\Delta_k = \sigma^2 (1 - 2^{2H-2}) / 2^{2kH} \tag{5.7}$$

We also deduce a similar equation from Eq. (5.2) by afBm parameters:

$$\Delta_k = \sigma^2 \left[f(2^{-k}) - \tfrac{1}{4} f(2^{-k+1}) \right] \tag{5.8}$$

N_{gauss} is a Gaussian random variable with zero mean and unit variance. Since the values of the Gaussian random variable mainly fall in some region, to estimate the span of the generated height values, we preset the probability value a to estimate the distribution region of this Gaussian random. As shown in Fig. 5.6(b), if the probability is that many random values are located in the shadow region that is a, the distribution region $[-G_0, G_0]$ can be deduced by:

$$P\{|x| < G_0\}=a \quad \rightarrow \quad P\{x < G_0\}=(a+1)/2 \tag{5.9}$$

where P is the probability of a set of random values and x is the Gaussian random variable. For example, if a is equal to 95%, the region in which most noise values are located is about $[-2, 2]$. We can control the span of the generated height values with a. The max-min height values of the fractal-box are set to $\overline{F}_T \pm G_0 \sum_{k=0}^{k_{\max}} \Delta_k$, where $\overline{F}_T$ is the average height of T. By this assumption, all generated height values are guaranteed to be within the fractal-box.

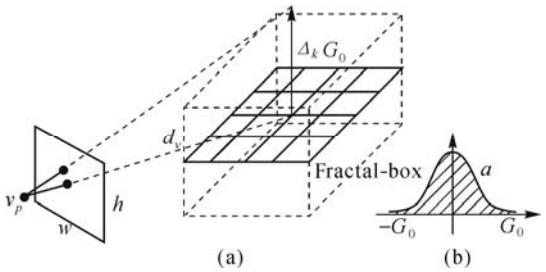

Fig. 5.6 Estimation of the projected error. (a) The 2D projection of the fractal-box; (b) The probability distribution

In the next step, the detail level of the to-be-generated fractal patch of T needs to be determined. The height values of the fractal patch are generated incrementally and the incremental step is $\Delta_k N_{\text{gauss}}$. As mentioned above, the step is less than $\Delta_k G_0$ when the probability a is given. Therefore, we project the height value onto the screen and compare it with the preset screen error tolerance E_{scr}. Let w and h be the width and the height of the viewport respectively, and fov be the field-of-view of the camera for rendering. The value of d_v is the minimum distance between the terrain region center of node T and the current viewpoint, as shown in Fig. 5.6. The projected error Prj_T is calculated by the following Eq. (5.10):

$$Prj_T = \frac{\max(w,h)}{2\tan(fov/2)} \cdot \frac{\Delta_k G_0}{d_v} < E_{scr} \tag{5.10}$$

If the current viewpoint is located in the fractal-box, the fractal patch with the maximum detail level, namely $k_{\max}$-th level, will be generated. Otherwise, the

level k is tested according to Eq. (5.10). If level k does not stratify the rendering requirements, level (k+1) is to be tested. The children of T are to be visited if level (k+1) exceeds k_{max}.

5.2.3.3 Five-Queue Optimal Refinement

During rendering, there exists frame-to-frame coherence for view-dependent meshes. Therefore, we need not traverse the afBm-tree in top-down order to select the afBm nodes for each frame. Instead, we can use the results from the previous frame as the starting point for the current frame to get the nodes for rendering to accelerate this procedure.

Unlike the dual-queue optimization (Duchaineau *et al*., 1997), we adopt five queues to maintain the nodes and generate the fractal patches, namely Q_r, Q_p, Q_g, Q_d and Q_m. Q_r records the afBm nodes used for rendering and Q_p keeps all the parents of the nodes in Q_r. The nodes in Q_g generate the new fractal patches with finer detail levels, and the nodes in Q_d abandon the current fractal patches and use the coarser fractal patches. Because the fractal patches on the coarser detail levels are cached, they need not be regenerated. Q_m records the parent nodes whose four children can be merged. Q_g and Q_m need to be sorted with priorities while the other three queues do not.

At the beginning of the rendering stage, the root node of the afBm-tree is sent into Q_r and the other four queues are empty. For each frame, firstly the priorities for all nodes in the five queues are updated and the nodes are moved to the appropriate queues. Secondly, the node T with the highest priority is selected from Q_g and the detail level index k of the to-be-generated fractal patch is determined by Eq. (5.10). If $k \leq k_{max}$, T will be moved into Q_r; otherwise, T will be split into four children and moved into Q_p. Its children are processed in the same manner. Thirdly, the node T with the lowest priority is selected from Q_m and we can check whether all children of T are in Q_r or Q_d. If so, the children are merged. Otherwise, its children in Q_d decrease their fractal patch levels and are moved into Q_r. The nodes in Q_g and Q_m are dealt with alternately until the nodes in Q_r reach the desired rendering accuracy.

The fractal patch produced by each node of Q_r may have different resolutions. T-vertices will occur at the contiguous afBm nodes if their generation levels are different.

We use a forced split approach (Pajarola, 1998) to solve the T-vertices. Given one node T in Q_r, its neighbor nodes may be forced to generate new fractal patches (Fig. 5.7(a)). This procedure may be iterative. As is illustrated in Fig. 5.7(b), where T is located at the left-bottom corner, after the right upper neighbor is forced to generate new fractal patches, their neighbors are also forced to do so. If the generation level of T is greater than k_{max}, T is split into four children. The splitting result of the left node (Fig. 5.7(a)) is illustrated in Fig. 5.7(c). Its neighbor

nodes may also be split too. When the left-bottom node (Fig. 5.7(b)) continues to generate, the right upper neighbor is forced to split into children (Fig. 5.7(d)). After the force procedure is finished, the appropriate connecting templates are selected to avoid *T*-vertices.

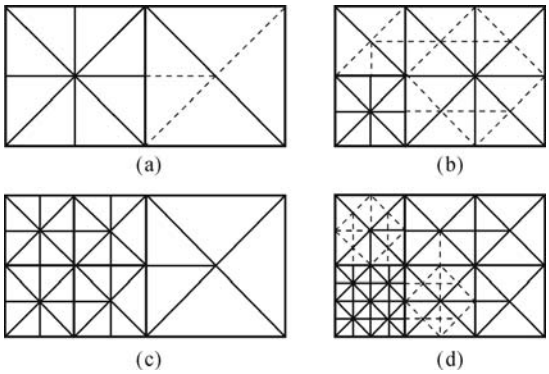

Fig. 5.7 Neighbor nodes are forced to generate new fractal patches or split according to the adjoining edges. (a) (b) (c) and (d) show the iterative generation of the fractal patches

5.2.3.4 Fractal Patch Generation by GPU

To utilize frame-to-frame coherence, we generate each fractal patch according to its detail level incrementally. This procedure is implemented on the GPU for acceleration.

The afBm parameters and the previous fractal patches are loaded into the video memory. If the output fractal patches were read back into the host memory, the transferring cost would be even higher than the generation cost. Fortunately, textures are allowed to be visited in Vertex Shader (VS) for current graphics hardware. Therefore, we keep the generated fractal patches as textures in video memory, which are to be retrieved by VS.

If the afBm parameters of the adjacent nodes are different, the values at the boundaries of fractal patches will be different and a crack artifact will occur, as shown in Fig. 5.8(a). We solve this problem by using afBm parameters based on vertices. For each vertex in the fractal patch, the afBm parameters of the adjacent nodes are interpolated by weights of the distances between this vertex and the centers of the adjacent nodes. The interpolated value is used as the afBm parameter of this vertex. The afBm parameters and the height values of all vertices are formed into a texture image and sent to the GPU. The result is illustrated in Fig. 5.8(b).

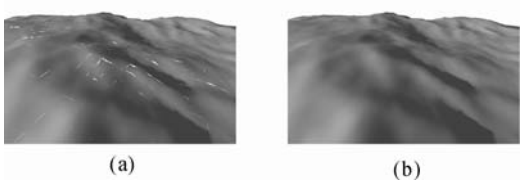

Fig. 5.8 Solving the crack artifacts. (a) Crack artifact occurs at the boundaries of the fractal patches; (b) Artifact is eliminated by use of the afBm parameters based on vertices

The fractal patches are produced in two passes: the generation pass in Pixel Shader (PS) and the rendering pass in VS. To generate the fractal patch of (k+1)-th detail level, three textures are required, as illustrated in Fig. 5.9. The first texture is the fractal patch in k-th detail level (Fig. 5.9(a)), which is used to calculate the average height values of fractal patches of (k+1)-th detail level. Bilinear interpolation is used here to compute the average values. We only need to set the texture magnification mode to GL_LINEAR in OpenGL to automatically accomplish this computation. Since we only need to generate height values for new vertices in fractal patches of level (k+1), the height values of existing vertices can be retrieved from the fractal patches of level k. The second texture, a mask map, is used to retrieve the old height values (Fig. 5.9(b)). If the pixel value is 1 in the mask map, the new height value needs to be calculated at this position; if it is 0, the value is copied from the patch of the level k.

The third texture is a pre-computed Gaussian map to make the value disturbance. According to the span probability a, an array of Gaussian noise is calculated before the rendering stage and stored as 2D texture (Fig. 5.9(c)). The coordinates of the terrain vertices are used as indices to look up the noise values in the Gaussian map.

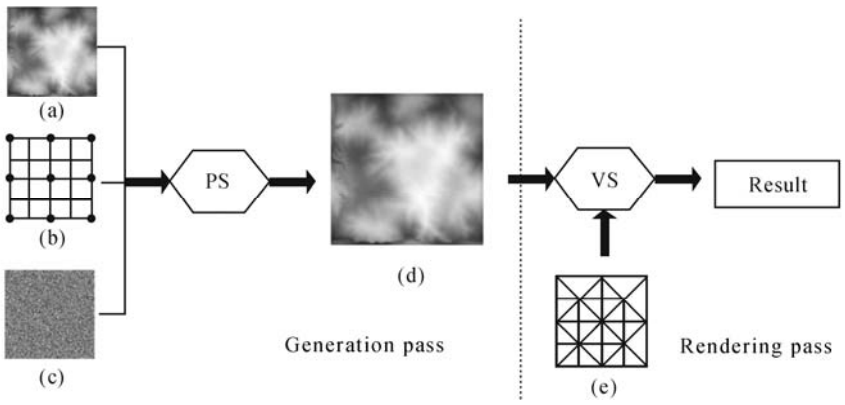

Fig. 5.9 The pipeline of the generation pass and the rendering pass. (a) The fractal patch of the k-th detail level; (b) The mask map for k-th detail level; (c) The Gaussian noise map; (d) All three textures are sent to PS to generate the new fractal patch of the level (k+1); (e) The rendering pass. The fractal patch and the connecting template are sent to VS to construct the terrain mesh for final rendering

We have implemented two kinds of generation approaches based on afBm: midpoint displacement (MD) (Fournier *et al*., 1982) and summing band-limited noise (SBLN) (Musgrave *et al*., 1989). The MD approach is faster than the SBLN one, but the generation quality is not as good as that of SBLN. In the MD approach, we use $\bar{F} + \Delta_k \cdot N_{\text{gauss}}$ to calculate the new height values, where $\bar{F}$ is the average value of its neighbor's height values and Δ_k is the same as that in Eq. (5.8).

The SBLN approach is actually a kind of spectral synthesis, which is the sum of a group of noise functions of increasing frequencies and decreasing amplitudes. The height value of vertex $v_{x,y}$ is $\bar{F} + \sigma^2 \sum_{i=0}^{octave} w^{-iH} n(w^i v_{x,y})$, where w is the lacunarity and usually set to 2. Octave is a constant, which controls the counts of the summing functions. The function $n(t)$ is the noise base function and we use the Perlin function based on Gaussian noise here.

After the fractal patches are generated, they are used together with the connecting template in the rendering pass. Height values in the fractal patch can be retrieved in VS in texture mode. In addition, the appropriate connecting template is passed to the GPU to construct the actual terrain mesh (Fig. 5.9(e)).

5.2.3.5 Texturing

We use two types of texturing techniques to improve the rendering quality. One is the pre-processing texture quad-tree and the other is procedural texturing. In the texture quad-tree, we build a texture quad-tree in the pre-processing stage given a huge texture. In the rendering stage, the nodes of the texture quad-tree are transferred from external storage to the memory (Cline and Egbert, 1998).

The terrain texture can also be obtained procedurally in the rendering stage. The material type map, including grass, sand and snow, is generated in the pre-processing stage. The type map and different material maps are sent to the GPU to compute the color of the terrain by multiple texture mapping.

5.2.4 Application

5.2.4.1 Terrain Data Compression

The afBm-tree model can be used to compress the terrain data to achieve a high compression ratio, which is beneficial for data transferring through the network. This approach is a lossy compression method. The compression error should be

properly controlled so that the results are rendered with required quality. The differences between the original height values of terrain vertices and the generated ones are calculated according to the following Eq. (5.11):

$$Dif = \frac{1}{n_g}\sum_{i=0}^{n_g}\frac{\left|F_0(v_i) - F_\Omega(v_i)\right|}{F_0(v_i)} \tag{5.11}$$

where the function F_0 represents the original height field and F_Ω is the generated height field under the threshold Ω. The value of n_g is the number of vertices on the generated terrain mesh. The greater the threshold Ω, the greater the compressed ratio. But the rendering quality will become worse. Therefore, we should set the appropriate threshold Ω to achieve a good balance between the compressed ratio and rendering quality.

5.2.4.2 Terrain Synthesis

It is difficult to edit the terrain to design new terrain features because of the huge size of the large-scale terrain data. For example, if we would like to synthesize a new large-scale terrain from the original terrain sample, the patch-matching process would be very time-consuming because of the huge data size. On the contrary, the size of the data in our afBm-tree structure is much smaller than the original data. The terrain can be synthesized in several seconds and the synthesized afBm-tree can be used directly for rendering, which is beneficial to designers.

We have borrowed the texture synthesis idea to synthesize new afBm-trees from existing ones. Here we adopt the patched-based texture synthesis method (Liang *et al*., 2001). Given one afBm-tree (Fig. 5.10(a)), we first convert it into a Mipmap structure to fit it for the patch-matching operation (Fig. 5.10(b)). An afBm-subtree is extracted in bottom-up order as the first patch, illustrated by the shadowed frames in Fig. 5.10(b). Then, several subtrees of the same level are selected from the given afBm-tree according to the fractal parameters of the first subtree, and one of them is selected randomly (Fig. 5.10(c)). The parameters at their overlapped region are recalculated by smooth blending, which is done for each level of the subtree. The selection and blending procedures are repeated until the new afBm-tree is completed (Fig. 5.10(d)).

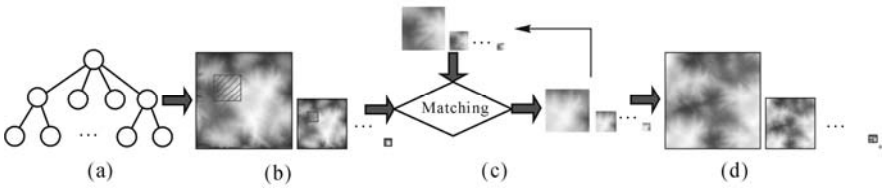

Fig. 5.10 The procedure of terrain synthesis with afBm-tree structure. (a) an afBm-tree; (b) the Mipmap structure; (c) patch-matching operation; (d) the synthesis result

5.3 Conclusion

To optimally represent and render large scale terrains, we propose an approximate model to emphasize the whole effect of natural scenes. Although such a model has certain differences from the original one, the human eye always attends to the whole effect instead of local details during the walkthrough, in wide-field natural scenes.

The terrain model is based on an afBm-tree. We can extract terrain meshes of different resolutions from this model according to given thresholds. Such approximation of original terrain data provides a high data compression ratio. It can also be used to produce view-dependent dynamic terrain meshes during rendering to avoid the time-consuming procedure of generating huge terrain models. The creation of terrain is accelerated by the GPU, which effectively reduces the amount of data transferred between the main memory and the video memory.

References

Bajaj CL, Pascucci V, Thompson D, Zhang XY (1999) Parallel Accelerated Isocontouring for Out-of-Core Visualization. Proceedings of the 1999 IEEE Parallel Visualization and Graphics Symposium, 97-104

Bishop L, Eberly D, Whitted T, Finch M, Shantz M (1998) Designing a PC Game Engine. IEEE Computer Graphics and Applications 18(1): 46-53

Boer WH (2000) Fast Terrain Rendering using Geometrical Mipmapping. http://www.connectii.net/emersion

Cignoni P, Ganovelli F, Gobbetti E, Marton F, Ponchio F, Scopigno R (2003a) BDAM-Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization. Computer Graphics Forum 22(3):505-514

Cignoni P, Ganovelli F, Gobbetti E, Marton F, Ponchio F, Scopigno R (2003b) Planet-sized Batched Dynamic Adaptive Meshes (P-BDAM). Proceedings of IEEE Visualization, 19-24

Cline D, Egbert P (1998) Interactive Display of Very Large Textures. Proceedings IEEE Visualization, 343-350

Dachsbacher C, Stamminger M (2004) Rendering Procedural Terrain by Geometry Image Warping. Eurographics Symposium Rendering

Duchaineau MA, Wolinsky M, Sigeti DE, Miller MC, Aldrich C, Mineev-Weinstein MB (1997) ROAMing Terrain: Real-time Optimally Adapting Meshes. Proceedings of IEEE Visualization, 81-88

Ebert DS, Musgrave FK, Peachey D, Perlin K, Worley S (1998) Texturing & Modeling. A Procedural Approach

Evans F, Skiena SS, Varshney A (1996) Optimizing Triangle Strips for Fast Rendering. Proceedings of IEEE Visualization, 319-326

Evans W, Kirkpatrick D, Townsend G (1997) Right Triangulated Irregular Networks. Technical Report, University of Arizona, 97-109

Fournier A, Fussell D, Carpenter L (1982) Computer Rendering of Stochastic Models, Communications of the ACM, 25(6): 371-384

Gerstner T (1999) Multi-resolution Compression and Visualization of Global Topographic Data. SFB 256 report 29, Univ. Bonn

Gotsman C, Rabinovitch B (1997) Visualization of Large Terrains in Resource-Limited Computing Environments. Proceedings of IEEE Visualization, 95-102

Gu X, Gortler SJ, Hoppe H (2002) Geometry Images. ACM Transactions on Graphics, Proc. SIGGRAPH, 21(3): 355-361

Herzen BV, Barr AH (1987) Accurate Triangulations of Deformed, Intersecting Surfaces. Computer Graphics, 21(4): 103-110

Hoppe H (1997) View-Dependent Refinement of Progressive Meshes. Proceedings of SIGGRAPH, 189-198

Hoppe H (1998) Smooth View-Dependent Level-of-Detail Control and Its Application to Terrain Rendering. Proceedings of IEEE Visualization, 35-42

Kaplan LM, Kuo CJ (1996) An Improved Method for 2D Self-Similar Image Synthesis. IEEE Transactions on Image Processing, 5(5): 754-761

Levenberg J (2002) Fast View-Dependent Level-of-Detail Rendering Using Cached Geometry. Proceedings of IEEE Visualization, 259-266

Lewis JP (1987) Generalized Stochastic Subdivision. ACM Transactions on Graphics, 6(3): 167-190

Liang L, Liu C, Xu Y, Guo B, Shum HY (2001) Real-Time Texture Synthesis By Patch-Based Sampling. ACM Transactions on Graphics, 20(3): 127-150

Lindstrom P, Koller D, Ribarsky W, Hodges LF, Faust N, Turner G (1996) Real-Time, Continuous Level of Detail Rendering of Height Fields. Proceedings of Siggraph, 109-118

Lindstrom P, Koller D, Ribarsky W (1997) An Integrated Global GIS and Visual Simulation System, Real-Time Optimally Adapting Meshes. Proceedings of IEEE Visualization, 81-88

Lindstrom P, Pascucci V (2001) Visualization of Large Terrains Made Easy. Proceedings of IEEE Visualization, 363-370

Lindstrom P, Pascucci V (2002) Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization. IEEE Transaction on Visualization and Computer Graphics, 8(3): 239-254

Losasso F, Hoppe H (2004) Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids. Proceedings of Siggraph, 769-776

Mandelbrot BB (1975) On the geometry of homogeneous turbulence, with stress on the fractal dimension of iso-surfaces of scalars. Journal of Fluid Mechanics, 72(2): 401-416

Musgrave FK, Kolb CE, Mace RS (1989) The Synthesis and Rendering of Eroded Fractal Terrains. Computer Graphics 23(3): 41-50

Nagashima K (1997) Computer Generation of Eroded Valley and Mountain Terrains. The Visual Computer, 13: 456-464

Pajarola RB (1998) Large Scale Terrain Visualization Using the Restricted

Quadtree Triangulation. Proceedings of IEEE Visualization, 19-26

Pascucci V (2000) Multi-Resolution Indexing for Hierarchical Out-of-Core Traversal of Rectilinear Grids. Proceedings of NSF/DoE Lake Tahoe Workshop, Hierarchical Approximation and Geometrical Methods for Scientific Visualization, 1-5

Pi X, Yang X, li S, Song J (2005) Patch-LOD Algorithm of Terrain Rendering Based on Index Template. Journal of Computer Research and Development, l42: 183-187

Platings M, Day AM (2004) Compression of Large-Scale Terrain Data for Real-Time Visualization Using a Tiled Quad Tree. Computer Graphics Forum, 23(4): 741-759

Pomeranz A (2000) ROAM Using Triangle Clusters (RUSTiC). Master's Thesis, U.C. Davis, CS Dept.

Pouderoux J, Marvie JE (2005) Adaptive Streaming and Rendering of Large Terrains Using Strip Masks. Proceedings of ACM GRAPHITE, 299-306

Rottger S, Ertl T (2001) Hardware Accelerated Terrain Rendering by Adaptive Slicing. Proceedings of Workshop on Vision, Modelling and Visualization, 159-168

Szeliski R, Terzopoulos D (1989) From Splines to Fractals. Computer Graphics 23(3): 51-60

Toledo R, Gattass M, Velho L (2001) QLOD: A Data Structure for Interactive Terrain Visualization. Technical Report, VISGRAF Laboratory

Ulrich T (2002) Rendering Massive Terrains Using Chunked Level of Detail Control. Course Notes of ACM SIGGRAPH

Vitter JS (2001) External Memory Algorithms and Data Structures: Dealing with Massive Data. ACM Computing Surveys 33(2): 209-271

Wagner D (2004) Terrain Geomorphing in the Vertex Shader. In: Shader X2: Shader Programming Tips & Tricks with DirectX 9. Wordware Publishing

6

Variational OBB-Tree Approximation for Solid Objects

Bounding volume hierarchies approximate complex objects with simple-shaped bounding volumes. For many time-critical applications, such as collision detection and real-time rendering, the bounding volume hierarchy is used instead of the original geometry to simplify the related computation, e.g. the intersection queries.

People have done a lot of work on how to construct bounding volume hierarchies. According to different applications, we can choose bounding volumes of different basic shapes to construct the optimal approximation hierarchies. These shapes include spheres(Bradshaw and O'Sullivan, 2004; Wang *et al*., 2006), axial-aligned bounding boxes (AABB) (Prusinkiewicz *et al*., 2001), oriented bounding boxes (OBB) (Gottschalk *et al*., 1996, Krishnan *et al*., 1998) and discrete oriented polytopes (k-DOPs) (Klosowski, *et al*., 1998). In this chapter, we introduce the variational Object Oriented Bounding Box Tree (OBB-Tree) which approximates a solid triangular mesh with minimal summed volume outside the mesh.

When we choose the basic shapes of bounding volumes we have to balance between the fitting tightness of the bounding volume and its simplicity. Although complex bounding volumes, such as k-DOPs, bind* the object more tightly, they require more complex intersection computations. On the other hand, efficient computation can be achieved with simple bounding volumes, but they bind* the objects less tightly. Weghorst *et al.* proposed a formula to evaluate the quality of bounding volume hierarchies in(Weghorst *et al*., 1984). This formula has been widely used in many applications involved bounding volume hierarchies, such as collision detection (Gottschalk *et al*., 1996). The evaluation function is defined as:

$$e = n_v \cdot c_v + n_p \cdot c_p \tag{6.1}$$

here n_v is the number for computing the intersection with bounding volumes, c_v is the cost of computing such an intersection once, n_p is the number of computer intersections with mesh faces and c_p is the cost of computing such an intersection once.

There have been different approaches proposed to construct bounding volume hierarchies aiming at different applications. For collision detection, the OBB tree proposed by Gottschalk is one of the most popular bounding volume hierarchies (Weghorst *et al*., 1984). In this section, we introduce the algorithm to construct the optimal OBB tree based on the data structure in (Weghorst *et al*., 1984). For the bounding box, c_v and c_p are constants in Eq. (6.1). Hence, to minimize Eq. (6.1), we only need to find the bounding box that has minimal n_v and n_p.

The better the OBB tree approximates the original object, the smaller are the number of intersections with original object's faces n_p. So we propose to use the outside volume as the measure of errors to evaluate how the OBB tree approximates the original object. Here the outside volume means the spatial parts of the bounding volumes that do not belong to the original object. This error does not contain the overlap of bounding boxes inside of the object. It only concerns the outside part of the bounding boxes. We can get the optimal OBB tree by minimizing this error. In this section, we will introduce the definition of outside volume, how to compute it and how to convert the problem of constructing an OBB tree to a variational approximation problem.

As for the minimal n_v, we need a tighter upper-level bounding box, since with a tighter upper-level bounding box we can quickly remove those bounding boxes of the next level that do need not to compute the intersection. Traditionally, people use a greedy method to construct the OBB tree in a bottom-up or top-down manner. However, since there is no effective measure to control the hierarchies of the bounding boxes and their tightness, it is hard to guarantee that the final OBB tree is optimal. Based on the outside volume, we adopt the multiple iterative optimization method similar to MultiGrid (Briggs *et al*., 2000) to optimize the bounding box hierarchies. Meanwhile, we adopt the clustering method of Lloyd (Lloyd, 1982) to optimize the bounding boxes of the same level. Based on these optimizations, we get the optimal OBB tree with global minimal error.

The following sections are organized as follows. We first introduce related work in Section 6.1. Then we define the approximation problem of OBB tree construction in Section 6.2. In Section 6.3 we give the optimization solver for the defined problem. And we show the experiments and results in Section 6.4. Finally, we give a conclusion in Section 6.5.

6.1 Related Work

The initial method for constructing the hierarchical bounding volume is to perform octree-subdivision to a certain depth in the world space (Hubbard, 1993; Liu *et al*., 1988). The original object is approximated by all non-empty leaf nodes containing mesh faces. Obviously, such a method which performs uniform-subdivision in the world-space has a worse bounding effect. Later people proposed Bounding Volume Hierarchies which constructed an hierarchical bounding volume in the

object space. Since a sphere has simple parameters, there are many algorithms which adopt spheres to construct bounding volume hierarchies (Bradshaw and O'Sullivan, 2004, Wang *et al*., 2006). However, such algorithms usually cannot bind* the object very tightly. The AABB-based bounding volume hierarchies are very fast to be computed (Barequet *et al*., 1996, Bergen, 1997), yet with not so good a bounding effect. The OBB-based bounding volume hierarchies can bind* the objects more tightly than AABB or sphere based bounding volume hierarchies. Besides, they usually have much less bounding volumes. However, it takes more time to compute the intersection for OBB than for AABB or spheres. K-DOPs are regarded as an extension of OBB. Although they bind* the object more tightly, the intersection computation is obviously much more complex for K-DOPs.

In this chapter we use OBB to construct bounding volume hierarchies. Existing methods mainly adopt top-down or bottom-up strategies to construct bounding hierarchies. Top-down subdivision methods do not use approximation degree as subdivision guidelines. Instead they use the longest axis of bounding volumes (Hubbard, 1995) or perform iterative subdivision with analysis of eigenvalues which are obtained through principle analysis to bounding volumes' spatial locations. Bottom-up methods only locally optimize approximation during combination (Gottschalk *et al*., 1996). Therefore, such methods usually get local optimal results with greedy algorithms.

Cohen-Steiner D *et al.* proposed a variational approximation method in (Cohen-Steiner *et al*., 2004) to approximate mesh objects. They defined the approximation problem as a variational energy optimization problem and got satisfying results. Wu J *et al.* extend the method in (Wu and Kobbelt, 2005) to use the high-order polynomial to approximate the mesh model in (Wu and Kobbelt, 2005). We introduced the variational approximation method to sphere approximation in (Wang *et al*., 2006) and got satisfying results. Lu *et al.* further extend the variational method to a bounding ellipsoid in (Lu *et al*., 2007). In this chapter we use OBB which has a more complex shape than a bounding sphere or ellipsoid. We cannot use the analytical form for spheres in (Wang *et al*., 2006) to compute OBB. In the following sections we describe our new method to construct an OBB tree.

6.2 The Approximation Problem of an OBB Tree

Here we use X to represent the 3D mesh, Y to represent the OBB tree. For an OBB tree with hierarchies, we use O_{ij} to represent an oriented bounding box j in level i of Y, and $Z_i=\{O_{i0}, O_{i1}, \ldots, O_{in}\}$ to represent the set of bounding boxes in level i of Y. When we discuss the bounding boxes of the same level, we omit the subscriber i and use $Z=\{O_j\}$ to represent all oriented bounding boxes of the same level. Since we focus on solid mesh models here, we use $\Omega(\cdot)$ to represent the space volume of the 3D object. An OBB tree Y of the 3D mesh X can be defined as:

Definition 1 Y is regarded as an OBB tree approximation of triangular mesh X, when Y satisfies $\forall x \in \Omega(X), x \in \Omega(Y)$.

Here x is one 3D point in the space. $Y = \bigcup_i Z_i = \bigcup_{i,j} O_{ij}$ is the union of a series of oriented bounding boxes with hierarchies.

To compute the optimal OBB tree, we define the error as the volume in Y that does not belong to X. Such a volume is called the Outside Mesh Volume, which is represented by $OMV(X,Y)$,

$$\begin{aligned} OMV(X,Y) &= \int_{R^3} g(x,X,Y)\mathrm{d}x \\ g(x,X,Y) &= \begin{cases} 1 & x \notin \Omega(X) \text{ and } x \in \Omega(Y) \\ 0 & \text{others} \end{cases} \end{aligned} \tag{6.2}$$

It is difficult to compute such outside mesh volume quickly and accurately for complex mesh models. We first introduce the outside mesh volume of the single oriented bounding box O for the mesh model X. Then the outside mesh volume of the oriented bounding boxes of the same level in the hierarchical OBB tree is given. Finally, we give the method for computing the outside mesh volume of the OBB tree.

We use $OBV(X,O)$ to represent the outside bounding object volume of the single oriented bounding box O. Similar to Eq. (6.2), we have

$$\begin{aligned} OBV(X,O) &= \int_{R^3} g(x,X,O)\mathrm{d}x \\ g(x,X,Y) &= \begin{cases} 1 & x \notin \Omega(X) \text{ and } x \in \Omega(O) \\ 0 & \text{others} \end{cases} \end{aligned} \tag{6.3}$$

According to Eq.(6.3), the outside volume of the bounding boxes of the same level $Z=\{O_0, O_1,\ldots, O_n\}$ for the original mesh model X can be represented as:

$$\begin{aligned} OMV(X,Z) &= OMV(X, \bigcup_j O_j) \\ &= \sum_j OBV(X,O_j) - \mathrm{Overlap}(OBV(X, \bigcup_j O_j)) \end{aligned} \tag{6.4}$$

In Eq. (6.4), $\mathrm{Overlap}(OBV(X, \bigcup_j O_j))$ represents the overlap parts in the outside bounding box volumes. We notice that the overlap parts are always positive, which means $\mathrm{Overlap}(OBV(X, \bigcup_j O_j)) \geq 0$. This leads to $\sum_j OBV(X,O_j) \geq OMV(X,Z)$, which means that $\sum_j OBV(X,O_j)$ can be used as the upper limit of the exact outside

bounding box volume $OMV(X,Z)$. Thus, as the value of $\sum_j OBV(X, O_j)$ decreases, the exact outside bounding box volume would decrease accordingly. Compared with computing the exact outside bounding volume, it is much easier to compute $\sum_j OBV(X, O_j)$. Therefore, in our implementation, we accelerate the computing by using the sum of all outside bounding box volumes in Z.

$$OMV(X, \bigcup_j O_j) \approx \sum_j OBV(X, O_j) \tag{6.5}$$

Since the tighter upper-level bounding boxes can effectively reduce the intersection computation for lower-level bounding boxes, it is more important that the upper-level bounding boxes approximate the mesh better. We introduce level-related weights to define the outside mesh volume of the OBB tree.

$$e = \sum_{i=1}^{n_v} \omega_i(i) \cdot OMV(X, Z_i) \tag{6.6}$$

In Eq.(6.6), $\omega_i(i)$ is the weighted function of the level i. The upper level bounding boxes have larger weights.

By introducing the outside bounding box volume, the construction of the OBB tree of the triangular mesh models is treated as a variational optimization problem:

$$\min e = \sum_{i,j} \omega_j(i) \cdot OBV(X, O_{ij}) = \sum_{i,j} \omega_i(i) \cdot \int_{R^3} g(x, X, O_{ij}) \mathrm{d}x \tag{6.7}$$

To solve this problem, we need to find the optimal bounding box approximation Y that makes the value of Eq.(6.7) minimal.

6.3 Solver for OBB Tree

6.3.1 Computation of Outside Volume for Single Bounding Box

Since mesh models are complex, we compute Eq. (6.3) in a discrete way. First, we consider the situation where the center of the bounding box is inside the mesh model X. As shown in Fig. 6.1(a), a ray l is radiated from the center of the bounding box in direction ω. The intersection points of the ray l and the mesh models X are recorded as p_0, p_1, …, p_n. The ray intersects the boundary of the

bounding box to get the last intersection point p_e. The part l_{out} outside the mesh model X of line section op_e is computed according to the following equation:

$$l_{\text{out}}(\omega) = \overline{p_e o} - \sum_i sign(n(p_i) \cdot \omega) \overline{p_i o} \tag{6.8}$$

where *sign* is the signed function. The contribution of each line segment to outside volume can be determined by the angle between the normal direction of p_i and the ray l. For example, in Fig. 6.1(a) the length of line outside the mesh (drawn in green) is computed as $l_{\text{out}} = p_e o - (p_2 o - p_1 o + p_0 o)$.

The whole outside volume can be obtained by integrating all directions in ω under local coordinates at the center of the bounding box:

$$OBV(X, O) = \int_{\Omega} l_{\text{out}}(\omega) \mathrm{d}\omega \tag{6.9}$$

When the center of the bounding box is outside the mesh model, we adopt the same method to compute the volume inside the mesh model (shown as the green line segment in Fig. 6.1(b)) and get the outside volume by subtracting this inside volume from the bounding box's volume.

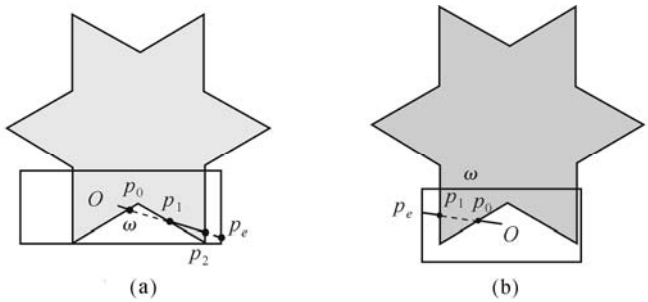

Fig. 6.1 The diagram showing how to compute the outside volume of the bounding box. (a) The center of the bounding box is inside the model; (b) The center of the bounding box is outside the model

In implementation, we accelerate computation with GPU. The center of the bounding box is used as the viewpoint and the 3D space is mapped to a CubeMap of resolution $6 \times m \times m$ (*i.e.* m=32). Each pixel on the face of CubeMap corresponds to a directional cone and each face of CubeMap is used as the projection plane. By rendering the bounding box O and mesh model X, we compute the volumes inside and outside the mesh of the directional cone through a shader. Finally, we get the outside volume by adding or subtracting these volumes.

We adopt four steps to compute the outside volume:

(1) Render the triangular mesh T_O of bounding box. Compute the depth value of intersection point p_e between the pixel direction $l(\omega)$ and T_O. According to this depth value, we compute the volume of the cone in the direction of $p_e o$.

(2) Render the triangular mesh T_X of the mesh model X, and send this model together with the depth texture in each pixel direction to the rendering pipeline. Compute the depth and the normal of the intersection point p_i between the ray and T_X by utilizing a vertex shader and fragment shader. Compare the depth of p_i and the depth of p_e stored in the texture in the fragment shader. If the depth of p_i is smaller than that of p_e, compute the cone volume in the corresponding direction. Otherwise, do not compute this volume. By utilizing the alpha blending in OpenGL, we add the volumes of directional cones for all intersection points and output this as the texture.

(3) Align the textures obtained from step 1 and step 2 and render them. Use a fragment shader to perform subtraction and get the final outside volume in each direction. Output this outside volume as the texture.

(4) Read the outside volume from the display card and perform a summation to get the whole outside volume.

6.3.2 Solver for OBB Tree

6.3.2.1 Solver for OBB of the Same Level

It is difficult to directly compute the OBB tree for complex 3D models. Cohen-Steiner *et al.* proposed a new approximation method in (Cohen-Steiner *et al*., 2004), in which they treat this problem as a global optimization for mesh segmentation. Based on their work, we propose the variational approximation algorithm for OBB trees of mesh models.

As for all faces F of the triangular mesh X, we find the segmentation of F, $\{F_0, F_1, \ldots, F_N\}$, which satisfies $F = \bigcup_i F_i$ and $F_i \bigcap F_j = \varnothing,\ i \neq j$. Such segmentation makes $\sum_i OBV(X, O_i)$ and the outside volume of bounding box O_i of F_i, has minimal value, which is:

$$\begin{aligned} &\min_{F_0, F_1, \ldots, F_n} \arg \quad e(X, Z) = \sum_j OBV(X, O_j) \\ &\text{s.t. } F = \bigcup_i F_i; F_i \bigcap F_j = \varnothing, i \neq j; F_i \in O_i \end{aligned} \tag{6.10}$$

We adopt the iterative Lloyd clustering algorithm (Lloyd, 1982) to solve the segmentation in Eq. (6.10). The flowchart of the algorithm is shown in Fig. 6.2. When the location and directional axis of the bounding box are fixed, the shape parameters of the bounding box are also determined for the fixed face subdivision. So we only need to optimize the six parameters related to the location and directional axis of the bounding box.

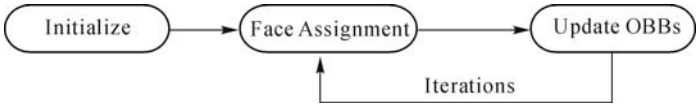

Fig. 6.2 Algorithm of OBBs approximation

(1) Initialization: For each bounding box, we select one or a few faces as seeds, randomly before subdivision. Fig. 6.3(b) illustrates the seeds to start with.

(2) Iterative Optimization: Our iteration includes 3 steps, which are to compute a new subdivision according to current bounding box parameters (center and directional axis), to update the bounding box parameters according to the new subdivision, and to adjust the subdivision to avoid the local minimal when needed.

(3) Face Assignment: The center of the bounding box remains unchanged during face assignment. We adopt the flooding algorithm which maintains a priority queue that stores the distance from each face to the nearest bounding box. At the beginning, for each center of the bounding box, select the nearest face as the seed and push it into the queue. During each iteration, we take the face $t_{\min}$ from the queue which has the smallest distance value to the center of the nearest bounding box, then push its neighboring faces into the queue. We assign the face $t_{\min}$ to a different bounding box O_i, and compute the change in its outside volume $dOBV(X,O_i)$ caused by the change in O_i'shape parameters from S_i to S_i'.

$$dOBV(X,O_i) = \| OBV(X,O_i(S_i')) - OBV(X,O_i(S_i)) \|$$

Our principal is to assign t_{min} to the bounding box O_k that has the smallest outside volume change, which satisfies: $\forall i \in \{1,2,...,N\}, dOBV(X,O_k) \le dOBV(X,O_i)$.

In implementation, we maintain a list for each bounding box which records the shape parameters and corresponding outside volumes. Such a list greatly simplifies the assignment computation and enhances the computation.

(4) The Update of Bounding Box Parameters: After the face assignment, for each bounding box there are face clusters F_i. We need to find the parameters for an optimal oriented bounding box for F_i that has minimal outside volume. Since it is very difficult to solve this problem analytically, we adopt a numerically discrete method. First we obtain the gradients by disturbance around the initial parameters. Then the optimal bounding box parameters are computed by the method of steepest descent. Since we use GPU to accelerate the computation of the outside volume under certain parameters, this discrete method can quickly update the bounding box parameters.

(5) Bounding Box Adjustments: Some initial values would make the above iteration become trapped in the local minimal. When the whole outside volume changes very little for some iterations, we sort the importance of bounding boxes according to the number of faces in each bounding box and the area of the overlap region among bounding boxes. By adjusting the least important one, we get a new approximation of the OBB. If the new approximation error is smaller than the

error before adjustment, we start iteration from the adjusted parameters. Otherwise, we reject this adjustment and quit the optimization. By incorporating the bounding box, our method effectively avoids the local minimal to get better results.

Fig. 6.3 illustrates such an optimization process. It can be observed that after iterative optimization, these resulting two bounding boxes are tighter than initial two.

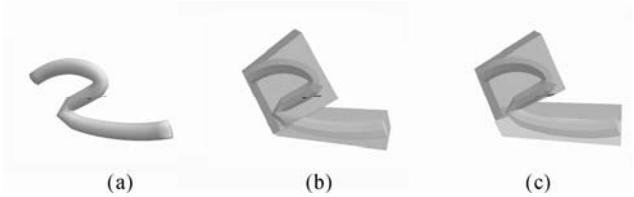

Fig. 6.3 Optimization process. (a) Bounding Box Adjustments; (b) Before optimization; (c) After iterative optimization

6.3.2.2 Solver for OBB Tree

Based on the method for computing the OBB of the same level and the algorithm of MultiGrid (Briggs *et al.*, 2000) to compute the multi-variable differential problem, we propose the multi-resolution iterative algorithm for the OBB tree.

This algorithm is shown in Fig. 6.4. By repeated iteration between different levels, we solve the approximation of an OBB tree that has minimal errors (Eq. (6.6)).

Fig. 6.4 Optimization of hierarchical OBBs

(1) Top-Down Decomposition Optimization: The decomposition number is given in advance for the hierarchical bounding box. For bounding box O_i with given face assignment, we use the method given in the last section to compute the bounding box approximation of the next level. The top-down optimization is changed to bottom-up optimization after the decomposition achieves the pre-defined recursive depth.

(2) Bottom-Up Combination Optimization: During combination, we start from the bounding box which is treated as the current leaf node. Level by level we perform combination optimization by the algorithm which is similar to the one used to get the bounding box approximation in the above paragraph. However, we do not use a mesh face as the subdivision unit. Instead we use a bounding box as the subdivision unit. The bounding boxes are combined by computing the optimized subdivision of these bounding boxes. After each combination, we compute the error of the combined hierarchical structure according to Eq. (6.6). If the error is larger than the value before combination, we turn to top-down

decomposition optimization instead of combination optimization. If the error is smaller, we continue to perform combination optimization for upper nodes.

Traditional optimization methods for computing hierarchical OBB usually adopt three kinds of methods, which perform optimizations in the bottom-up way, in the top-down way (Barequet *et al*., 1996; Goldsmith and Salmon, 1987; Klosowski *et al*., 1998; Omohundro, 1989), or by inserting nodes gradually. These traditional methods are mainly greedy strategies which only consider the optimization for the current level and do not optimize the whole error. Compared with these traditional methods, our method defines the whole error and manages to get the smallest whole error by iterative optimization. Since our method needs to iterate between different levels, it costs more time compared with traditional methods.

6.4 Experiments and Results

The experimental results were obtained on the computer with the configuration of Intel 2.8 GHz CPU, 1 GMB memory and nVidia Geforce 7900 GT display card.

We choose to construct the binary OBB-tree. The weight value between levels is set to be $\omega_i=l_{sub}$, where l_{sub} is the number of levels from the current node to the leaf node. For example, the leaf node has $\omega_i=1.0$ and the node just above the leaf node has $\omega_i=2.0$. In our experiments, we get satisfying results with such values of ω_i.

Although we adopt some acceleration algorithms, our method still needs more time to construct an OBB-tree compared with traditional methods. In our experiments, it costs 20~30 ms to compute the outside volume for one OBB with GPU. Since a large amount of computation to compute outside volumes is required in face subdivision and the update of bounding box parameters, it costs 2 to 3 h to construct a binary OBB tree with 12 levels which has 2^{12} leaf nodes.

Since the method in (Gottschalk *et al*., 1996) is the most widely used algorithm to compute an OBB-tree and the source code RAPID is provided (RAPID), we compare our algorithm with this method in terms of the bounding box's visual appearance, geometry error, and the computation efficiency in collision detection.

In order to test the efficiency of approximation results in real collision detection, we build the collision detection test environment in Fig. 6.5 based on an open-source physical engine ODE (ODE, 2007). In this environment, objects continually fall down and rotate from the top. We perform the intersection test between the falling objects and the plane. If the falling objects intersect with others, the objects will be bounced off. During collision detection, we respectively perform the intersection computation with the approximation OBB-trees produced by our method and the method in (Gottschalk *et al*., 1996). We record the number of times we compute intersections between a falling object and the bounding box in unit time to compare efficiency. The less the time needed to compute intersections, the better the OBB-tree approximates to the model. For comparison, we test the collision detection over a long period and choose a lot of (>1000) objects falling down

stochastically. The result of the intersection number/time is shown in Table 6.1, where we can see that our OBB-tree is better than the one in (Gottschalk *et al*., 1996) for a lot of collision detections. For three test models in our experiments, we can reduce the intersection computation by 8% to 20%.

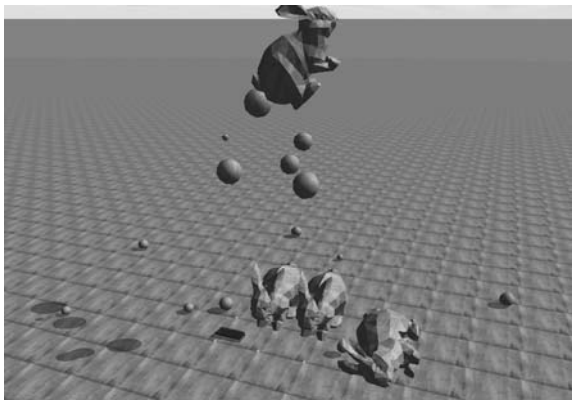

Fig. 6.5 The collision detection test environment

Table 6.1 The comparisons of the computing efficiency of two hierarchical OBBs in the collision detection test environment.

	The percentage of the number of intersections that have been reduced
Snakie	19.85%
Armadillo	12.36%
Dinosaur	9.05%
Bunny	14.52%
Turtle	16.81%

6.5 Conclusion

In this chapter we introduce a new method to construct the OBB-tree of solid mesh models. We propose to use the outside volume as the measurement of how the bounding box approximates the mesh model and transfer the OBB-tree construction to a variational approximation problem. In order to get the optimized OBB-tree, we adopt the multi-level iterative algorithm similar to MultiGrid (Briggs *et al*., 2000) and the Lloyd clustering method in (Lloyd, 1982) to compute the global optimized result so as to obtain the tightest approximation result. As shown in the experiments and comparisons, our method is better than previous methods in terms of geometry error and intersection computation in collision detection.

The most obvious drawback of this method is its long computation time. Compared with traditional methods, our method is more suitable for applications where higher-quality approximation results are required and the preprocessing time is relatively long. Since our method uses outside volume as error measurement, we can only deal with the closed mesh model which has spatial volume. However, in real applications there are a lot of non-closed mesh models. So we will further work on how to compute the OBB-tree for more general mesh models.

References

Barequet G, Chazelle B, Guibas LJ, Mitchell JSB, Tal A (1996) BOXTREE: A Hierarchical Representation for Surfaces in 3D. EUROGRAPHICS '96 Computer Graphics Forum, Blackwell Synergy, 387-396

Bergen GVD (1997) Efficient collision detection of complex deformable models using AABB trees. J. Graph. Tools, 2(4): 1-13

Bradshaw G, O'Sullivan C (2004) Adaptive medial-axis approximation for sphere-tree construction. ACM Trans. Graph., 23(1): 1-26

Briggs WL, Henson VE, McCormick SFA (2000) Multigrid tutorial: second edition. Society for Industrial and Applied Mathematics, Philadelphia

Cohen-Steiner D, Alliez P, Desbrun M (2004) Variational shape approximation. ACM Press, New York, 905-914

Goldsmith J, Salmon J (1987) Automatic creation of object hierarchies for ray tracing. IEEE Comput. Graph. Appl., 7(5): 14-20

Gottschalk S, Lin MC, Manocha D (1996) OBBTree: A Hierarchical Structure for Rapid Interference Detection. In: Proceedings of ACM SIGGRAPH, 171-180

Hubbard P (1995) Collision Detection for Interactive Graphics Applications, Brown University Providence, RI, USA

Hubbard P (1993) Interactive collision detection. In: Proceedings of the 1993 IEEE Symposium on Research Frontiers in Virtual Reality, 24-31

Klosowski JT, Held M, Mitchell J, Sowizral H, Zikan K (1998) Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs. IEEE Transactions on Visualization and Computer Graphics, 4(1): 21-36

Krishnan S, Pattekar A, Lin M, Manocha D (1998) Spherical shells: A higher-order bounding volume for fast proximity queries. In: Proceedings of the 1998 Workshop on the Algorithmic Foundations of Robotics, 122-136

Liu Y, Noborio J, Arimoto S (1988) Hierarchical sphere model HSM and its application for checking an interference between moving robots. In: Proceedings of the IEEE International Workshop on Intelligent Robots and Systems, 801-806

Lloyd S (1982) Least squares quantization in PCM. IEEE Transactions on Information Theory, IT, 28(2): 129-137

Lu L, Choi YK, Wang W, Kim MS (2007) Variational 3D Shape Segmentation for

Bounding Volume Computation. Computer Graphics Forum, 26(3): 329-338
ODE (2007) http://www.ode.org/
Omohundro SM (1989) Five Balltree Construction Algorithms, ICSI Technical Report tr-89-063, University of California, Berkeley
Prusinkiewicz P, Mundermann L, Karwowski R, Lane B (2001) The use of positional information in the modeling of plants. In: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, ACM Press, 289-300
RAPID. http://www.cs.unc.edu/~geom/OBB/OBBT.html
Wang R, Zhou K, Snyder J, Liu X, Bao H, Peng Q, Guo B (2006) Variational sphere set approximation for solid objects. The Visual Computer, 22(9): 612-621
Weghorst H, Hooper G, Greenberg DP (1984) Improved Computational Methods for Ray Tracing. ACM Trans. Graph., 3(1): 52-69
Wu J, Kobbelt L (2005) Structure Recovery via Hybrid Variational Surface Approximation. Computer Graphics Forum, 24(3): 277-284

Index

A
AABB, 31, 281, 283, 292
ACE, 235, 242
afBm, 265, 266, 268-273, 275
afBm-tree, 266, 268-270, 272, 275-277
appearance part, 2
Ashikhmin-Shirley Reflectance, 17
asymptotic fBm,265, 266

B
blocking service request, 235, 238, 243, 244
bounding box hierarchy, 27
bounding volumes, 281-183
BRDF, 13, 15-19
built-in visitors, 114

C
CABTT, 260, 261
Cached Aggregated Binary Triangle Trees, 260
CAppearance, 55, 212, 213
CBasicVisitor, 109-112
CBillboard, 33-35, 81, 111, 171, 175, 210
CBone, 39
CBoundedFeatureVisitor, 111, 112
CBSpline, 55
CBufferGPUResource, 126
CCamera, 116, 121, 123
CCameraManager, 123, 133
CColor, 44, 85
CColorRGBA, 44, 85
CComposedCubeMapTexture, 60, 61
CComposedRigidMotion, 39
CComposedShape, 33, 36, 37, 81, 171, 173
CCoordinate, 43, 85
CCubeMapTextureGPUResource, 125
CFeatureTypeClassifier, 169, 170,187
CGeoOrigin, 45, 51
CGPUResourceManipulator, 124, 128, 129
CGroup, 32, 33, 83, 111
CIE, 14, 15
CImage, 59, 85
CImageTexture, 58, 215
CImageTextureGPUResource, 125, 130
CIndexedTriangleSet, 42, 43, 45, 46
CKD2TreeComposite, 76
CKD2TreeIterator, 105
client-server system, 195, 196
CLightCullingPreRender, 173, 174, 187
CLoadingThread, 165, 166, 168
CMaterial, 57, 213
CMirrorPreRender, 163, 164
CMLine, 55
CMText, 55
CMultiTexture, 61-63, 65, 67, 70
CMultiTextureCoordinate, 43, 81
CMultiTextureTransform, 67

CNormal, 43, 85
control node, 195-197
COoCEntityLoader, 165, 166, 169
Cook-Torrance Reflectance, 17
CPoint, 55
CPolyline, 55
CPreRenderManager, 137, 140, 141, 160
CRelation_SISG, 74, 76, 77, 81, 82, 114, 163
CRenderingEngine, 115-117, 119, 122, 123, 134, 139, 186, 187
CRenderModuleManager, 159, 160
CRenderQueue, 121, 123, 140
CRenderQueueElementProcessor, 171, 176, 177
CRenderQueueManager, 123
CRenderServerState, 240, 242
CRenderTarget, 121, 131, 132
CSceneGraph, 74, 92, 96, 204
CSceneGraphIterator, 100-105
CSceneModelImp1, 82, 91, 92, 94, 95
CSceneModelManipulator2,94, 201-203
CSceneStorageImp1, 91-94
CScreenState, 240
CServerManager, 233, 236
CServiceRequestManager, 234, 236, 237
CServiceRequestSender, 238
CServiceRequestSRHandler, 242
CServiceRequestSRThreadPool, 242
CServiceRequestTranslator, 235, 238
CSGComposite, 73, 74, 209
CSGDynLeaf, 74, 109, 208, 209
CSGLayer, 74, 81
CSGLeaf, 73, 74, 76, 77, 79-81, 101, 109, 208, 209
CSGSwitch, 74, 209
CShader, 68
CShaderProgram, 68
CShape, 33-37, 42, 81, 111, 171, 175, 176, 187, 210
CShapeRenderModule, 174-177, 186
CShapeRenderPipeline, 174, 175
CShapeRMAttribute, 176, 177
CSkeleton, 39
CSkinAnim, 38, 111, 215
CSkinShape, 33, 38-40
CSpatialIndexComposite, 76
CSpatialIndexLeaf, 76, 77, 109
CSphereMapTexture, 60, 61
CStaticLOD, 37, 76, 81, 111, 171, 173
CSuperShape, 33, 35-37, 40, 78, 79, 81, 111, 171, 176, 210, 211
CText, 55
CTextureBackground, 40
CTextureCoordinate, 43, 85
CTextureCoordinate3D, 43, 85
CTextureTransform, 66
CTransformGroup, 33
CTriangleMesh, 47
CTripleIndex, 45
CVFCullingIterator, 162-163
CVFCullingPreRender, 161, 164, 169, 174, 187
CViewpoint, 41
CWinIno, 116

D
digital elevation models, 263
display environment configuration, 6
display space, 200, 218
display space frame, 220, 221

E
Entity, 2, 3, 22, 26, 28, 82-85, 87, 89-91, 93, 95-98, 165, 168, 170, 171, 199
event handler, 101, 102, 110, 112, 145, 161, 163, 167, 170

F
fBm, 263, 265, 266, 269, 271
fractional Brownian motion, 263
fragment shader, 72, 185, 287

G
garbage collection, 83, 84, 89, 90

GC, 50, 51, 53,
GD, 50-53, 55
geometric part, 2
Geometrical MipMap, 261
Geometry Clipmap, 262, 264
geometry processing phase, 10
GeoMipMap, 261
global coordinates, 11
global space, 11, 27
graphic primitives, 193-196
grouping relationship, 2

I
IAnimatedFeature, 31, 32, 38, 111, 198, 201
IAppearanceChildFeature, 55, 56
IAttributedObject, 29, 30, 79
IBindableFeature, 32, 40
IBoundedObject, 31, 32, 74, 76, 111, 112
IChildFeature, 31, 37, 173
IColorFeature, 44
IComposedGeometryFeature, 42, 45, 46
ICoordinateFeature, 43
ICurveFeature, 42, 55
IDynamicObject, 74
IEntity, 28-30, 73, 76, 107, 108, 110, 122
IEnvrionmentTextureFeature, 58
IFeature, 29, 30, 32, 74, 76, 82, 111, 129, 130, 152, 155
IGeometryFeature, 34, 42
IGPUResource, 124, 128, 129, 130
IGroupingFeature, 31, 32, 37, 111
IImageFeature, 58-60
IIndexFeature, 44, 45
ILightFeature, 32, 40, 70, 198, 201
IMaterialFeature, 55, 57
IModularRenderPipeline, 147, 148, 150, 151, 155, 175
IMotionFeature, 39, 74, 75
INormalFeature, 43
INSUP, 251, 252, 254, 255
IOutOfCoreObject, 85, 165, 168
IPreRender, 121, 122, 137-140, 142, 145, 187
IRenderControlUnit, 134-137, 139, 140, 145
IRenderModule, 157, 159
IRenderModuleAttribute, 149, 159, 152, 155
IRenderPipeline, 121, 122, 142, 143, 145, 147, 165, 187
Irradiance, 14, 15
ISceneIterator, 99, 100, 102
ISceneModel, 82, 86, 91-93, 95-97
ISceneStorage, 88, 89, 91-93, 95
IServiceRequest, 234, 237, 243, 244
ISGNode, 73, 74, 82
IShaderFeature, 56, 67, 68, 186
IShapeFeature, 31-33, 37, 39, 70, 111, 198, 201
ITextFeature, 42, 55
ITexture2DFeature, 58
ITextureCoordinateFeature, 43
ITextureFeature, 58, 62
ITextureTransformFeature, 66, 67
IThreadSafeObject, 85, 93
IVisitor, 107-109, 110

L
Lafortune Reflectance, 16
Lambertian Reflectance,16
level-of-detail, 4, 278
lightmapping, 5
loader, 3, 165

M
master-slave system, 196
MD, 275
midpoint displacement, 275
model coordinates, 11
model space, 11
model transforms, 11
modular render pipeline, 147, 152, 157-159, 174
multichannel display, 193

multiple-field data types, 25

N
node field management, 3
node lifetime management, 3
node relation management, 3

O
OBB-tree, 281, 190-192
object table, 28, 83, 93
observer controller, 6
out-of-core entity, 85, 87, 93, 165, 168
out-of-core technique, 262, 263
outside bounding object volume, 284
Outside Mesh Volume, 284, 285

P
parallel rendering, 193-196, 256
persistent pointer, 26
Phong Reflectance, 16
pixel shader, 11, 274
polygonal mesh, 2
precomputed radiance transfer, 19, 20
procedural terrain rendering, 258, 263, 264, 268, 270

Q
Quadtree, 259-261, 279
query utility, 3

R
Radiance, 12-15, 18-20, 185
Radiosity, 5, 14
Real-time Optimally Adapting Meshes, 260, 277, 278
real-time rendering, 7, 9, 11, 19-21, 191, 258, 262, 281
relation schema, 76-80
Render Control Unit, 121, 134, 164
render node, 195-197
renderable object, 2, 4, 5, 198
rendering equation, 12, 13, 17, 18, 71, 72
rendering pipeline, 5, 7, 9-11, 66, 193, 194, 287
rendering process, 5, 10, 197
rendering protocol, 197
Restricted Quadtree Triangulation, 260
Right Triangulated Irregular Networks, 261, 278
ROAM, 260, 261, 279
RQT, 260
RSG, 258, 259
RTIN, 261

S
SBLN, 275
scan conversion phase, 10
scene graph, 2-5, 22, 26-28, 73, 74
scene manipulator, 93, 94
scene model, 1, 2, 21, 26, 37, 74, 76, 79, 80, 82-88, 90-95, 97099, 111, 117, 118, 135, 137, 142, 165, 190, 197, 198, 201, 202, 204, 209, 224, 233, 245
scene model management, 1, 21, 94, 98, 201
scene model schema, 74, 76, 79, 81
scene storage, 28, 82-84, 86, 89, 93, 94
serializer, 3
service callback function, 234
service request manager, 234-236
service request sender, 235, 239
service request translator, 235
shading, 5, 6, 10, 12, 15, 18, 19, 164, 185, 213, 214
shadows, 6, 18, 20
single-field basic data types, 22
sort-first system, 194, 195
sort-last system, 194
sort-middle system, 194
spatial index, 3, 28, 31, 75-79, 81-84, 87, 88, 94, 96, 99, 105, 109, 111, 117, 137, 207, 208
spatial relations, 3, 5, 6

special visual effects, 6
storage device, 83, 87, 89, 92-94
summing band-limited noise, 275
system state manager, 235, 236, 238

T
TCouple, 45
texture mapping, 5, 64, 65, 72, 182, 185, 275
TIN, 258, 259, 263
transform hierarchy, 27, 28
transformation, 2, 3, 18, 19, 27, 33, 39, 66, 67, 70, 73, 263
TTriple, 45

U
user input phase, 9, 10
UTM, 50, 51, 54, 55

V
VCRS, 196, 197, 201, 218, 219, 221, 222, 230, 231, 235, 250, 251
vertex shader, 10, 11, 69, 72, 273, 287
view frustum, 3, 27, 78, 161, 163, 164, 173, 187, 194, 195, 261
view object, 225
view transform, 11, 12
virtual local screen, 219, 231, 254
visiting mode, 84
VRE, 196-198, 203, 204, 208, 245
vsibility Culling, 4, 78, 174
vxCMessage, 220, 228
vxIAnimation, 198, 201, 215
vxIAppearance, 198, 212, 228
vxICamera, 198, 228
vxIEnvironment, 198, 201, 217
vxIGeometry, 198, 211
vxILight, 198, 201, 203, 216, 245
vxIModel, 198, 199, 201, 237
vxIRenderingSystem, 198, 199, 218, 230, 237
vxISceneGraph, 198, 199, 201, 203, 204
vxIScreen, 198, 220, 230
vxISGNode, 198, 199, 201, 204, 208, 209, 244
vxIShape, 198, 199, 201, 210, 211, 212, 245
vxITexture, 198, 215, 213
vxIUI, 198, 199, 218, 219
vxIUIControl, 198, 230
vxIViewport, 198, 224, 228
vxIWindow, 198, 222

W
Ward Reflectance, 16